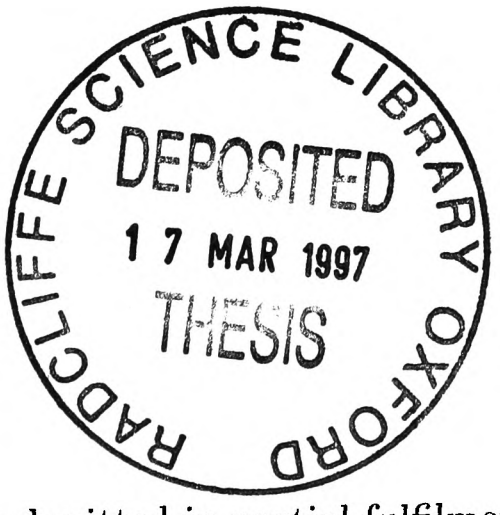
Design of a Hyper-Environment for Tracing Object-Oriented Requirements

Francisco de Assis Cartaxo Pinheiro

Wolfson College





A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at the University of Oxford, Michaelmas Term, 1997

Abstract

Change is inevitable and unending in developing large, complex systems. Changes to requirements arise not only from changes in the social context of the system, but also from improved understanding of constraints and tradeoffs as system development proceeds. *How to trace software requirements* is the problem addressed by this thesis. We present a solution for requirements tracing in the context of object-oriented software development. Our solution consists of a traceability model and a tool to automate the tracing.

TOOR, the tool to implement the model, uses a *project specification* written in FOOPS, a general purpose object-oriented language with specification capabilities, to set up the environment in which a project is carried out. The project specification defines the trace units and traces as objects and relations, respectively. The evolution of objects from requirements sources to requirements to design to code, and generally to any object taking part in the process is dealt with in a uniform way in TOOR: classes are declared for each kind of object we wish to control, and relations are defined between them.

TOOR uses regular expressions to provide a *selective tracing mode*: the actual configuration of objects and relations is considered as a text and regular expressions are used to retrieve parts of the configuration matching the pattern described by them. TOOR enhances the flexibility of regular expressions by extending the pattern matching procedure by providing different ways of specifying how an object or relation is to be matched. Other modes of tracing in TOOR are the *interactive tracing* through modules and the *non-guided tracing* through several browsing mechanisms. TOOR modules are used to structure projects by providing hierarchical scopes for objects used in a project development. The tracing mechanisms of TOOR can use the project structure to order searches or to provide boundaries for searching. Browsing objects provides additional flexibility in situations where little information of what has to be traced is possessed and hyper-media features address the need to re-interpret data usually encoded in different formats.

The user-definable features of a project specification provides much of the flexibility necessary for effective use of a software tracing tool. Also, the integration of regular expression tracing with other forms of tracing such as browsing and interactive tracing makes TOOR an extremely versatile tool. The user can select the more appropriate form of tracing depending on context and can switch from one form to another as convenient.

Acknowledgements

I am very grateful to my supervisor, Professor Joseph Goguen — so many good suggestions and valuable comments that I cannot account for. However, I would like to thank him not only for his technical guidance and encouragement throughout my research, but also for his personal attitude. In fact, in a Popperian sense, I can say he is the best *supervisor theory* that I know.

Regrettably Joseph had to leave before the completion of my thesis. But most fortunately Dr. Grant Malcolm took over his duties as my supervisor for the final months. His help was invaluable, giving me friendly support and technical encouragement in moments of great pressure. He deserves my thanks as both a supervisor and a friend.

My examiners, Prof. Anthony Finkelstein and Dr. Lincoln Wallen, provided me with valuable criticisms. The thesis is much improved for their comments; they gave me the opportunity to say things that I haven't said before.

The work on this thesis was supported by a research grant from CNPq (Brazilian National Council for the Development of Science and Technology) under a research grant number 200666/91-6.

My thanks to all those from our Requirements Group. In particular, Andre Stern, who read the first drafts of my work (an unenviable task!) and provided me with valuable criticisms. Marina Jirotka and Alan Munro were also important in the development of my research. They gave me great opportunity to discuss, and learn about, the social aspects of requirements.

My colleagues in the Programming Research Group, Răzvan, Hendrick, Paulo, José Barros, and Lutz deserve thanks as well as my roommates Dominic, Ranko, Ben, Corina, and specially Ana who also helped me with a lot of practical things.

Rui Camacho and Vick Joffe are great friends that I made here. They alone are worth my stay. My parents always deserve mention. I can feel their support in more than one way.

My deepest gratitude goes to those two who helped me walk through the tortuous path leading to this thesis: my wife and my son Francisco. I know that my wife Jane endured hard times waiting for me to finish this thesis, and I am very grateful for her tolerance. A million thanks are due to her, and I will never forget her support.

to my father, who couldn't see the outcome
of this challenge, and
to my son, who will see this one
and many more to come.

Contents

1 Introduction **1**

1.1 A Historical Perspective on Requirements Traceability 2

1.2 Requirements Traceability 4

1.3 Concepts for a Software Tracing Model 5

1.3.1 Trace Characteristics 6

1.3.2 Trace Definition 6

1.3.3 Trace Production 7

1.3.4 Trace Extraction 8

1.4 Aspects of Tracing 9

1.4.1 Functional Tracing 9

1.4.2 Nonfunctional Tracing 10

1.4.3 Pre-Requirements Specification Tracing 10

1.4.4 Environmental Tracing 10

1.5 A Case for (Yet Another) Traceability Support 11

1.6 Requirements for a Software Tracing Model 15

1.6.1 Similarities in Life Cycle Models 16

1.6.2 Critical Visions of Life Cycle Models 20

1.7 Overview of TOOR's Tracing Model 22

1.7.1 Defining Traces 23

1.7.2 Producing Traces 23

1.7.3 Extracting Trace Information 24

1.7.4 Functional, Nonfunctional, and Environmental Tracing 24

1.7.5 Architecture of the System 24

1.8 Preliminary Evaluation of the Solution 25

1.9 Thesis Contribution 26

1.10 Thesis Structure 28

2 Defining an Environment for Tracing **30**

2.1 Relations as Traces 31

2.2 Project Specification 32

2.2.1 Object and Relation Menus 33

2.2.2 Class Templates 34

2.2.3 Creating Objects 35

2.2.4 Creating Relations 35

2.3 Structure of a Project Specification 35

2.3.1 Project Specification Constraints 37

2.4 Standard Project Specification 38

2.4.1 Common Structure of Objects 40

2.4.2 Specifying Objects 41

2.4.3 Using External Files to Specify Attribute Values 42

2.4.4 Restricting the Value of Object Attributes 43

2.4.5 Using Axioms to Specify Attribute Values and Operations 44

2.4.6 Specifying Relations 46

2.4.7 Specifying Relation Attributes 48

2.4.8 Specifying Relation Axioms 49

2.4.9 Specifying Indirect Links 50

2.4.10 Automatic Relations 52

2.5 Project Evolution 52

2.5.1 Adding a new class 53

2.5.2 Changing classes 55

2.5.3 Removing classes 56

2.5.4 Changing relation axioms 56

2.5.5 General Modifications 57

2.5.6 Customizing the Interface 57

2.6 Discussion 58

3 Using Modules for Requirements Tracing **59**

3.1 TOOR Modules 62

3.1.1 Interactive Creation and Use of Modules 64

3.2 Module Syntax 65

3.2.1 Declaring Classes 65

3.2.2 Importing Modules 66

3.2.3 Transitive Module Importation 68

3.2.4 Selective Module Importation 69
3.3 Registering Objects in TOOR Modules 70
3.3.1 Declaring Registered Objects 71
3.4 Extended Example 72
3.4.1 The Development Environment 72
3.4.2 Eliciting Requirements 73
3.4.3 Specifying the System 76
3.4.4 Creating Nested Modules 78
3.5 Automatic Generation of Traces 80
3.5.1 Automatic Traces of Structural Information 80
3.5.2 Automatic Traces of Domain Information 82
3.5.3 Tracing Missing Links 84
3.5.4 Using Automatic Tracing Facilities to Make Informed Changes 85
3.6 Other Aspects of Using FOOPS in TOOR 87
3.6.1 Registering General FOOPS Texts 87
3.6.2 Using Previously Written FOOPS Files 88
3.6.3 Generating FOOPS Specifications 88
3.7 Tracing with Modules 89
3.7.1 Interactive Tracing 89
3.7.2 Background Tracing 90
3.7.3 Specifying a Starting Module 91
3.8 Using Modules in Other Trace Modes 92
3.9 A Note on Method 92
3.10 Discussion 94

4 Using Patterns to Trace Objects **95**
4.1 Tracing with Regular Expressions 96
4.1.1 Using the Tracing Template 97
4.1.2 Introductory Example 98
4.1.3 Searching for an Object 99
4.1.4 Searching a Sequence of Related Objects 99
4.1.5 Searching Alternative Patterns 99
4.1.6 Searching for Repeated Patterns 100
4.2 Overview of TOOR's Regular Expression Tracing Mechanism 101
4.3 Referring to Objects and Relations 104
4.3.1 Referring to Configuration States 105

4.3.2 String Conformance 108
4.4 Viewing Objects and Relations as a Text 110
4.5 Finding the Matching Strings 113
4.5.1 Presenting Regular Expression Tracing Results 115
4.6 Extending the Regular Expression Language 117
4.6.1 Regular Definition for Object Identifier 118
4.6.2 Regular Definition for Classes 118
4.6.3 Regular Definitions for Properties 119
4.6.4 Expressing Composition of Relations 121
4.6.5 Expressing Iteration of Relations and Objects 121
4.6.6 Expressing Exclusion of Objects and Relations 123
4.7 Restricting the Tracing Space 125
4.7.1 Using Modules to Restrict the Tracing Space 126
4.8 Discussion 127

5 Enhancing Traceability **128**
5.1 Browsing Objects 129
5.2 Using Hypermedia Features 131
5.2.1 Defining External Links 133
5.2.2 Tracing Contents of Attributes 134
5.3 Discussion 136

6 Tracing Requirements Evolution **137**
6.1 An Introductory Example 138
6.1.1 Registering Objects and Relations 138
6.1.2 Modifying Already Registered Objects by Edition 139
6.1.3 Modifying Already Registered Objects by Replacement 140
6.1.4 Evolving Object's Descriptions 141
6.1.5 Trying Alternative Developments and Abandoning Objects 142
6.2 Keeping Track of Evolution in TOOR 143
6.2.1 Deriving and Refining Requirements 144
6.2.2 Changing and Abandoning Requirements 146
6.2.3 Abandoning Objects 147
6.3 Tracing Object Modifications 148
6.4 Discussion 149

7 Tracing With Existing Domain Models **151**
7.1 Contributions Structures 151
7.2 gIBIS-like Models 154
7.2.1 Enlarging Models of Objects and Relations 157
7.3 Keeping Specific Features 159
7.4 Discussion 160

8 Comparisons **162**
8.1 TOOR summary 162
8.2 Academic Approaches 164
8.2.1 Trace Definition 165
8.2.2 Trace Production 167
8.2.3 Trace Extraction 169
8.3 Industrial Strength 171
8.3.1 RDD-100 172
8.3.2 RTM 173
8.3.3 DOORS 174
8.3.4 Discussion 174

9 Conclusions **184**
9.1 Future Work 184
9.2 Revisiting TOOR's Requirements 186
9.3 Evaluation 189
9.3.1 Validity 190
9.3.2 Requisites 191
9.3.3 Usability 192
9.3.4 Organizational Impact 193
9.3.5 Adaptability 194
9.3.6 Effectiveness and Scalability 196

A Formal Aspects of TOOR and Deferred Proofs **198**
A.1 TOOR Modules 198
A.1.1 Module Signatures 198
A.1.2 Module Structure 200
A.1.3 Using Objects 201
A.2 TOOR Grammar and Automata 202

A.3 Derivatives of Regular Expressions 203
A.4 Derivatives of TOOR Regular Definitions 204

Bibliography **207**

Chapter 1

Introduction

Tracing requirements is essential in developing large systems. Tracing helps in verifying system features against the requirements specification, in identifying error sources, and most significantly, in managing change. Requirements evolve throughout the lifetime of a system. They change in nature, from general to specific; they change in scope, by incorporating new features and deleting old ones; and they change in content and in form, as they are modified to become more consistent, precise and clear. They are also connected to each other and to other artifacts involved in system development. Moreover these connections also evolve over time.

In such a dynamic environment, aggravated by the existence of hundreds, or even thousands, of requirements, it is very hard to keep clear where one is in a project, especially for new people or people working in a new area of the system. Decisions about what should be modified in view of a (potential) change may require knowing the evolution of the parts under consideration and their connections. Moreover, the aspects relevant to such decisions are likely to vary depending on the context of the change and on the person interested in assessing the change.

How to trace software requirements is the problem addressed by this thesis. We present a solution for requirements tracing in the context of object-oriented software development. Our solution consists of a traceability model and a tool called TOOR (for Traceability of Object-Oriented Requirements) to automate the tracing. The tool is designed mainly as a research tool. It has a broad scope and its use is expected to improve the understanding of requirements related issues.

This chapter introduces both the problem and the solution in the following way. First we put the problem in context, giving a historical account of the research efforts related to requirements traceability. Then we state the problem in terms of its definition and discuss the concepts of trace definition, trace production, and trace extraction. This is followed by

an overview of TOOR, showing how it addresses these concepts. The general requirements of our traceability model are presented alongside with a discussion of software life-cycle models intended to provide a rationale for those requirements. Finally, we make a preliminary assessment of the solution, highlighting the achieved contributions and explain how the solution is developed in the remaining chapters.

1.1 A Historical Perspective on Requirements Traceability

The investigation of causes of failures in the development of large software systems soon identified the requirements area as a primary source for these problems. Requirements traceability was also identified as necessary to effectively manage requirements complexity and changes. In his paper on software engineering, Boehm [13] mentions traceability to design and code as a desirable property for requirements. Indeed, requirements traceability can be traced back to the early days of research on the field of software engineering at the end of the sixties and beginning of the seventies. Although, to our knowledge, there is no comprehensive historical account for this, we can find requirements traceability issues in several requirements related papers of that time. Two examples are the pioneer research projects called Information System Design and Optimization System (ISDOS) and Software Requirements Engineering Program (SREP).

In 1971, Teichroew and Sayani [137] reported on the development of ISDOS at the University of Michigan to improve the requirements and design phases of software development. It was one of the earliest efforts towards machine processable requirements. The Problem Statement Language (PSL) and the Problem Statement Analyser (PSA) [136] are direct products of this project; requirements traceability is not explicitly mentioned in the basic references on them, but their structuring capabilities and extensive information extraction facilities reflect a concern for the need for traceability support.

SREP was sponsored in 1973 by the Ballistic Missile Defense Advanced Technology Center (BMDATC) to address specifically the requirements phase of the development of large real-time systems. The program produced the System Requirements Engineering Methodology (SREM) [4, 7] consisting of a set of procedures, a System Requirements Language (SRL) [9], and a Requirements Engineering Validation System (REVS). In this case requirements traceability was a primary aspect of the project. One of its goals is stated as "to provide automated aids to keep track of requirements change and to report the interactive effects of requirements statements". Also the SRL language has explicit constructs to register traceability information.

Although ISDOS and SREP are pioneer works and representative of the initial research efforts in this period, requirements and requirements traceability concerns can be traced back even further. Davis and Vick [34] relate that the SREP program was motivated by the need to improve the development of large real-time ballistic missile defense systems. The first of these systems is reported to have begun in 1957. The sheer complexity of such systems and the need for reliable and precise requirements led to the establishment of several research programs prior to SREP. One of them, the System Technology Program [147], is said to have employed automated traceability aids.

There are reports on several other requirements approaches during the decade of 1970. Two examples are the Requirements Language Processor (RLP) [31, 26] developed at GTE laboratories as part of the Requirements Processing System (RPS) [27] and the Specification and Description Language (SDL) [125]. Requirements traceability is again not the primary topic in most of the approaches in this period, but it is frequently addressed in one way or another. For example, one of the major goals of RPS was stated as "to produce well-formatted document including table of contents and cross-reference tables which is suitable for customer inspection as well as for software design". Nevertheless, there are some specific works on requirements traceability. In 1977, Dorfman and Flynn [38] described ARTS, a tool designed specifically to trace requirements, and in 1978 Pierce [108] reported on a Requirements Traceability Tool that had been previously used in large Navy systems.

The early research efforts on requirements were not restricted to the United States of America. The EPOS [82] system is reported to have been initiated at the University of Stuttgart, Germany, in 1973. It underwent experimental use during 1978 and 1979 and it was industrially applied from 1980. Also, in 1971, Kawashima et al. [80] described the use of state transition diagrams to specify requirements in Japanese companies. Most of these early approaches took the form of a requirements language, sometimes used in connection with a database and a set of tools to evaluate and check for requirements properties. Requirements traceability was present but it was seldom the central focus. In these approaches, requirements traceability almost always shared the focus of attention with, and sometimes gave room to, other requirements issues such as preciseness of description, consistency, completeness, etc.

This trend of research went on up to the end of the eighties. This period is called by Greenspan et al. [67] the *first wave* of requirements research. The *second wave* begins in the nineties and is marked by the appearance of regular dedicated conferences on the area. Research may be characterized by greater attention to nonfunctional issues, by the development and use of more comprehensive domain models, and by an enlargement of the

view about requirements, now encompassing social aspects of software development. Research on requirements traceability also follows these lines and assumes a more prominent role.

1.2 Requirements Traceability

The understanding of requirements traceability evolved much along the lines described above: from a functional perspective where requirements relate to other artifacts according to some decomposition of the development process to a non-functional view where requirements relate also to things such as people, policies, etc.

There are several definitions for requirements traceability in the literature, each one emphasizing one or other aspect of what is supposed to be the ability to trace requirements. A standard definition is given below. It states the basic understanding of requirements traceability. A requirements specification document (and the requirements in it) should be referenced by the design documents and by other documents produced in the software development process. This allows for the evaluation of the effects of change or the reassessment of information used in software development. For the same reason, the source of each requirement should also be clearly indicated.

> **Traceability.** *A software requirements specification is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhanced documentation* (IEEE [100]).

Although this definition makes clear the need to relate requirements to other documents and to their sources, it does not quite capture the dynamics necessarily involved in the process. The second definition below is more recent and highlights that what has to be addressed is the evolution of requirements and not only snapshots of their descriptions. It encompass the dynamics of the process in the sentence "on-going refinement and iteration" which is explained as involving development, use, refinement, etc.

> **Requirements Traceability** *refers to the ability to describe and to follow the life of a requirement in both a forwards and backwards direction (i.e., from its origins, through its development and use, and through periods of on-going refinement and iteration in any of these phases)* (Gotel and Finkelstein [64]).

This definition is closer to our view. However, we should highlight a prominent aspect of requirements traceability: the natural occurrences of traces. We trace a requirement back to a document because the requirement carries traces of the document. It may be that the requirement is directly extracted from the document, it may be that the document contains

statements supporting the requirement, or it may be something else. In any case the existence of some *influence* of one on another is necessary if they are to be viewed as *related* objects. In the same way, we trace a requirement forward to a design module because the module carries traces of the requirement. Again, it may be that the module is designed to meet the requirement, it may be that the module is designed to specify a test procedure for the requirement, or it may be something else. In this view we have to identify which traces we want to control, how they may be captured, and how they may be followed afterwards. Our working definition will be

Definition 1 Requirements Traceability *refers to the ability to define, capture, and follow the traces left by requirements on elements of the software development environment and the traces left by those elements on requirements.* □

This definition depends on previous notions. For us, a software development environment involves not only the technical, but also the social aspects of software development. Its elements comprise not only the technical artifacts such as specifications, diagrams, and code, but also people, policies, decisions, and even less tangible things like goals and concepts.

The significant idea is that *traces are naturally produced* as a result of activities, actions, decisions, and events happening during software development. If a requirement is stated as a result of some discussion, traces of the discussion are present on the requirement at the very moment of requirements elaboration. This is just as footprints are present on a surface at the very moment someone walks on it. Therefore, to define what traces should be considered, what elements should be involved, how trace production may be captured, and how traces should be retrieved are central to any tracing model.

1.3 Concepts for a Software Tracing Model

We use the term *software tracing model* to encompass any model devised to trace software artifacts, not only requirements. According to our definition those models should define what *sorts* of traces can be modelled and should provide ways to capture and to retrieve them. The concepts of trace definition, trace capture, and trace extraction described below will guide the development of our solution. They also will be used later to justify and to argue for the adequacy of our model in dealing with the informal and situated aspects of software development.

1.3.1 Trace Characteristics

A trace is defined in ordinary usage as *a mark, track, sign, etc. showing what has existed or happened* [97]. The 'mark' does not need to be a material thing. It can be, for example, *a non-material indication or evidence of the presence or existence of something, or of a former event or condition.* In the natural world to trace something amounts to look for, or to follow the course, development, or history of traces, or occurrences of traces, left by whatever is being traced.

In computer science terms the picture is slightly different because this definition is blurred by the need for representation. Things should be defined and represented in machine processable ways in order to be accessible by computer programs. Moreover, whatever is defined in that way should also be produced in machine processable terms, i.e., it should be registered or stored in a medium that can be accessed by computer programs. Also, the ways of accessing those representations should be machine effective. Therefore any tracing model is circumscribed with respect to trace definition, trace production, and trace extraction.

1.3.2 Trace Definition

When one software object leaves traces on another software object, those objects may be viewed as related in some way. The nature of the trace is represented by the particular relation indicating how the two objects interact. For example, consider that a requirement is derived from another one indicating that although they may be different requirements one induced the existence of the other. If we trace this requirement back to the one from which it was derived then we may view the derived requirement as carrying traces of the other. The trace itself may not be apparent, it may not be possible to infer the trace only from the expression of the requirement, but it exists in the relation between the two requirements and it was produced at the moment of deriving one requirement from the other.

In order to be able to register traces of software objects a tracing model should define what its *trace units* are and what its *traces* are, i.e., how they are represented in the model. A tracing model should also define what *sorts* of traces may exist, indicating what the trace is intended to represent, what are the units that may be involved, and under what conditions traces may be registered.

Example 2 Consider the situation where a software system S has a source code module M. The question of what functionality the module is intended to provide could be answered, for instance, by "M is intended to provide the functionally described by requirement R". In that sense there is a trace of requirement R on module M. When the requirement was considered

in the design of the system, the requirement left a trace on it, otherwise the design could have been different. The first condition to register and retrieve that trace is the existence of some representation of S, M, and R. □

Example 3 Consider the situation where someone is interested in verifying if an object was kicked or held by someone else. In this case footprints and fingerprints, as traces of human contact, are quite different and have a definite role in the interpretation of the contact. On the other hand, if the only interest is to verify if there was some human contact, fingerprints and footprints are alike and both serve the purpose just as well. □

A tracing model may define just one sort of trace to cover all possibilities, but this would leave the interpretation of the trace to take place outside the model. For example, a tracing model may have only one kind of link to indicate the there is a trace of an object on another one. However, as all links are alike it is not possible to determine from the information in the model what are the different types of traces. Only in very specific cases it is possible to fully define a trace so that its interpretation would be formally precise. Nevertheless, even if the interpretation is left to be done outside the model, it is still possible to incorporate in the model indications of how the trace should be interpreted. For example, one may use named links to give additional information of what the trace is meant to represent.

1.3.3 Trace Production

In the natural world a trace is produced as a result of actions, events, or conditions which naturally impress or leave a mark of their effect. Software traces are also naturally produced as a result of the working practices of software development. However, the use of automated tracing models may not capture traces as they are produced.

Example 4 Consider the situation where an analyst is conducting an interview and designing diagrams based on the information he receives. The diagrams are produced in accordance with the information received. In this sense it is possible to say that the interview leaves traces on the design. Moreover, these traces are naturally produced as a result of interviewing and designing. If the analyst uses a tracing model to register the design chart, the interview transcript, and the link between them after they have been produced, then the production of the trace is incorporated into the model but in an artificial way, as an after the fact event. □

The trace production itself can sometimes be automated. In this case the software development activities are carried out using tools that are integrated with the tracing model in a way that allows the automatic registration of traces.

Example 5 Consider the same situation as above. This time the analyst is using an automated design program and the interview is being recorded directly into a computer. Moreover, the design program and the recording apparatus are part of a wider software environment which automatically register the design chart and the sound file as trace units. In this situation the trace production may also be automatically registered. It may be considered natural since there is no need to produce the trace again, performing after the fact activities to register the trace units and their links. □

To register a trace is to document the trace units and the trace itself. The two ways of trace production just described are defined, with respect to the registration of traces, as

- **Off-line**, if the registration of traces and trace units is performed as an after the fact activity. The off-line production of traces may be *manual* or *automatic*. The second case may be exemplified by the use of procedures to automatically extract trace information from existing documents.

- **On-Line**, if the activities producing traces are automated in a way that the traces and trace units are registered as a result of performing them.

Trace production is an important aspect of tracing models not only because one can trace only what is available, but also because it may interfere directly with the activities of developing software. It may impose an overload on people carrying out these activities. The less intrusive the trace production, the more efficient and accurate the use of the tracing model is.

1.3.4 Trace Extraction

In order to follow a trace it is necessary to extract the registered representation of the trace. The trace may be extracted in several ways and the trace extraction features of a given tracing model depend on how the trace is defined and on how the trace is produced.

Trace extraction depends on trace definition and production to the extent that one can extract only what has been registered. Trace extraction also needs to be defined in the model in terms of an effective procedure to perform the extraction. A tracing model should provide different and flexible ways to extract information registered in it, so that the most appropriate one can be chosen for each occasion.

Example 6 Consider again the situation where a software system S has a source code module M as a result of the existence of requirement R. One way of extracting the trace of R on

M is simply to get the trace unit R, given M. Another way would be to get both units R and M and the links between them that indicate the existence (and maybe the sort) of the trace.

However, it may not be so clear how the requirement has influenced, or determined, the module. If the module is too complex, if it is intended to meet several requirements, if it is badly written, or for some other reason, it may be difficult to understand the relation between R and M. The question of how R came out to be related to M could be answered, for instance, as "R was split into requirements R_a and R_b; requirement R_a was allocated to design module M_a which was coded as module M". This still shows the trace of R on M but it also shows a history indicating how the trace was produced. □

1.4 Aspects of Tracing

The development of a system establishes the context where information is defined, produced, and used for tracing purposes. Trace information can be related to the *functional* and *non-functional* aspects of the system being developed, and to the the *social* or *environmental* aspects of the development process. A tracing model can also be seen under each one of these aspects as a way of understanding how the definition, production, and extraction of traces may be related to actual software development practices.

1.4.1 Functional Tracing

Functional tracing relates to the question of whether a given functionality is present in the system being delivered. It tracks the evolution of the functions and behaviour prescribed for the system, together with their associated data. It is associated with tracing functional requirements which are defined as those requirements stating the functions and the behaviour a system must perform or exhibit.

Functional tracing depends on the method used to develop software. One can represent in a tracing model only those artifacts that a software development method or technique being used is capable of producing. For example, from the use of entity-relationship diagrams to design systems it is possible to associate trace information to the concepts of an entity, a relationship, an entity attribute, etc., but not to the concept of data flow. Also, from the use of a waterfall life cycle model to develop a system, it is possible to associate trace information to a requirements specification document, to a design document, to a program, but not to a prototype. Functional tracing is the easiest aspect to automate. Given a well defined process or method for producing software artifacts, traces may be generated based on the relations determined by the process or method being used.

1.4.2 Nonfunctional Tracing

Nonfunctional tracing relates to the question of whether a system possesses a given overall attribute. It tracks the evolution of nonfunctional requirements together with their associated data.

In general, nonfunctional requirements are not allocated to particular design units but usually encompass several parts of the system. They may even apply to the whole system. Therefore, a tracing model should be able to define and to register the production of traces involving groups of trace units. Nonfunctional tracing does not depend entirely on the software development method being used. It uses trace units such as test procedures or evaluation points that may be applied to many methods or techniques.

1.4.3 Pre-Requirements Specification Tracing

Pre-Requirements Specification (pre-RS) traceability is a concept introduced by Gotel [64, 63]. Pre-RS tracing relates to the development of the requirements specification document itself and concerns the social aspects involved in its production. The trace information may not be related to any function or attribute of the system being delivered. Of more importance are the relations determined by the social environment and work practices of the people involved in producing requirements related artifacts. This is what is called in [63] *personnel-based* requirements traceability, in contrast with artifact-based requirements traceability which would be more relevant to the functional and nonfunctional aspects of tracing. A pre-RS trace is intended to register the relevant factors or influences contributing to the existence of requirements.

Although pre-RS tracing is related to the requirements phase, the production of a requirements specification document, it should be available throughout the system development. Frequently the need to trace some software artifact back to a requirement is followed by a need to go even further backwards.

Pre-RS tracing needs a model establishing the relevant trace units and the sorts of trace that can be related to these units. The simplest model consists of providing annotations for the documents used or produced during the requirements phase. More complex models for pre-RS tracing also pay attention to the social environment and to the way requirements related artifacts are produced. We discuss one such model later on.

1.4.4 Environmental Tracing

Environmental tracing is an extension of pre-RS tracing. It is also intended to capture traces that the social environment may leave on any other trace unit, but it views the traces produced

by the environment as happening at any time in the development process. In fact Gotel [63] suggests that the principles of pre-RS can be applied to later phases of the development.

The social environment is an integral part of any system development. It has several facets related to people, policies, culture, organization, etc. It has effects on the initial phases of the development process, particularly the requirements phase, but it also affects all other phases. Environmental tracing relates to the question of how a given aspect of the system is affected by the social environment in which its development is carried out. It tracks changes in the environment and their effects on the development process and on the system itself.

Environmental tracing needs a model of the environment in order to establish the relevant trace units and the sorts of traces that can be related to these units. Again, the simplest model is through textual annotations containing an informal description of what has been considered in the production of some software artifact. More complex models may involve, for example, concepts for scheduling of activities, allocation of personnel, department structure, and other management and organizational aspects.

1.5 A Case for (Yet Another) Traceability Support

The diversity and huge amount of information dealt with when developing large software systems points to the need for automated tools to support development practices, including traceability. It could be argued that requirements traceability is satisfactorily tackled by existing tools and environments to automate the development process. Thus, configuration management systems support the definition of project baselines and control of software versions, process centered environments allow the definition and control of software processes and the enactment of associated tools, CASE tools provide support for several phases of the development, etc. All these tools and traceability tools in particular already cover the existing traceability demands. The heart of such arguments is that existing automation provides the necessary and sufficient support for traceability insofar they allow the organization of information. The non-supported features are those not amenable to automation. Here, our position against this view will be of a general nature but enough to show that it is still possible to advance current traceability practices. Throughout this thesis we give specific examples extending the points made below.

The argument in favour of the adequacy of current traceability support may be described along two lines. A first line of argument simply asserts that current traceability support already enables the registration and retrieval of related information in a useful way. When developing large systems, it is hard to remember all links that were made to relate information and it may even be impossible, in a distributed or multi-team development, to know that

they exist. Moreover, the matter is complicated when immediate links to a given piece of information are not what is required. One usually wants to follow a chain of links that goes far away from the starting point of tracing. Current traceability support comes in handy for these cases, allowing the retrieval of important related information which would otherwise be difficult to find.

A second line of argument explains the reasons why current traceability support already provides sufficient automation. It is based on the fundamentally unstructured way in which information is gathered and used. Information itself is in many cases unstructured or, better saying, has a complex structure which is dependent on the (social) context. Moreover, the production and posterior use of this kind of information is also complex and context sensitive. Therefore, no pre-defined structure will contain the necessary elements to satisfy the real needs for traceability. The search for related information is guided by necessity not by pre-defined structures. As a result, the provision of a basic framework to relate information is what is required in practical terms. Needs beyond that may be dealt with by means of techniques to keep and retrieve unstructured information, like HTML documents and general text editor systems.

These arguments find a powerful support in the fact that increasingly large and complex software systems are constantly being built with varying degrees of success and traceability is somewhat achieved in the process. After all, for a system to be satisfactorily completed implies that people have performed the natural going back and forth of the development process. However, we suggest that the existing traceability support is not as adequate as it should be (or at least, as it can be). We argue that current traceability tools do not provide full support, and that traceability is achieved through personal contact and searches that go beyond automation, e.g., group discussions and inquiries. There is some evidence for this view, which explains the existence of many proposals to enhance traceability through models aimed at capturing requirements related discussion [119, 118, 121], design rationale [112], personnel based traceability [63, 65], etc. Besides viewing traceability as being sometimes realised beyond automation we argue for the advancement of current traceability support on the same grounds used to justify its adequacy.

Most of the time the information should be traced beforehand in order for it to be (automatically) traceable in future. This circularity is enforced by current traceability tools, which are based on the existence of explicit relations between objects (through links, attributes, etc.). To register related information in these tools amounts to establishing a trace; one links documents to requirements only if there is a perceived trace between them. In the same way requirements are linked to design modules and those to code and so on. As

one traces only what has been related, the net result is that problems that require forward or backward traceability to or from any given object are supported only to the extent that (traceability) links already exist. The same can be said of almost all traceability techniques: requirements are tagged with information known to be relevant; cross references are made with objects known to be related; traceability matrices are filled with known connections, etc. Therefore, the current support for traceability is restricted, to a great extent, to the extraction of information already made traceable.

However, tracing existing links can hardly be considered as satisfying all needs for tracing. Two problems can easily be identified. On one hand, in an environment characterized by dynamic change and in which the unfolding process of developing information is open to negotiation and to several solutions, it is not possible to foresee all links that are going to be needed in future. On the other hand, even for those links that may be thought of in advance, it may not be adequate to register them all. For this second point there is an interesting illustration. In a study for the NASA Goddard Space Flight Center Mission Operations and Data Systems Directorate (MO&DSD)[1] a commercially successful requirements tool is said not be used because people "felt they spent more time than they wanted on traceability". What is interesting is the fact that a requirements traceability tool is left out precisely because it requires too much traceability. In addition, an existing link may not give the necessary information to trigger the desire of a person to follow it in all relevant cases.

As for the impossibility of complete automation, we agree that much of the information used in system development, mainly in its initial phases, is of an unstructured nature or, at least, little is gained from imposing one particular structure. We believe that this is true not only of information viewed as a unit, but also of information viewed as relations. Although it is our view that tracing cannot be fully automated (e.g., in the extreme case it is even possible that the information wanted is not stored in a computer at all; which does not mean that such information was not used at some point in the development), we believe that the basic framework provided by existing traceability tools may be enhanced to cope with some unstructured information. Although the arguments may not be the same, our conclusions agree with the conclusions of a recent workshop on requirements engineering [111] which identified a need for improvements in tool support for traceability to automatically record the traces and provide suitable interfaces for using trace information, and in tool support for structuring and documenting requirements as well as support for several kinds of

[1]The report is part of the MO&DSD tools capability inventory and is available on the net http://joy.gsfc.nasa.gov/MSEE/doors.htm. It only reflects a personal judgement but illustrates our point that traceability information is not always dealt with in advance. A more thorough assessment of DOORS is given in section 8.3.3.

representation such as video, text, graphics, diagrams, etc.

This thesis describes an abstract traceability model that should be instantiated with concrete models to be operational. It may be viewed as consisting of three languages; to specify the concrete traceability models, to express traceability patterns of related objects, and to define the module structure in which a project is carried out. In addition to its many distinguishing features our model address the two points discussed above: it makes possible from inside a single environment to trace information according to relations not thought of in advance, and it provides structuring facilities that can be made as general or specific as convenient. First, relations in our model may be specified by axioms and two objects may be viewed as related as a result of evaluating the axioms; they do not need to be explicitly related. Also, the value of objects' attributes is specified using abstract data types that may be implemented in different ways. Therefore, we may have methods to access the contents of external files given as attributes' values. These methods may be used to trace objects, searching for information which were not necessarily used when relations were made.

The traceability structure of our model is given by a formal specification defining the objects and relations. It provides a wide range of structuring facilities. At one end the traceability structure may be comprised just by identifiers linked by simple links, and at the other end it may be comprised by complex objects connected by an also complex network of relationships. The objects' attributes may refer to external files containing unstructured information but, at the same time, by providing suitable definition for their abstract data types, varying degrees of structure may be attributed to them. But not only objects and their relations may be provided with different degrees of structure. The development itself may be structured using modules and the tracing of objects may take into account the structure given for the development. Modules are not just a reference to development phases; they provide scopes where the actual development takes place. They may be composed and imported in several ways and, therefore, using them when tracing objects is more than just a documentational aid saying in which phase each object was created: modules are used to reflect the dynamic use and transformation of objects across all phases of the development.

These features are integrated in a tracing language used to express patterns of related objects. It is a regular expression based language with extended matching procedures. The language uses the actual specification provided for objects and relations and, therefore, methods to search external files, axioms constraining the use of objects and relations, and the actual contents of objects' attributes can all be used when expressing traceability queries. These features advance and provide additional support for requirements traceability making possible to define varying degrees of structure for the information to be traced and the tracing

of information not thought in advance.

One question that remains to be answered is to what extent traceability can be automated in a useful way. After all, design and requirements are recognized as 'wicked problems' [21, 148, 130] where the problem domain and the solution domain overlap and solution for one aspect of the problem usually results in subtle ramifications, unfolding a chain of previously unsuspected problems. In such an intangible and inherently inconsistent environment it may be argued that traceability will always be achieved through non-automated initiatives. It may be the case that, as Wieringa points out [146], "the world is the ultimate traceability tool". Despite the platonic connotations of such a statement, this view is in accordance with some of the traceability practices highlighted in the preceding paragraphs. According to our view, it is not possible to fully automate requirements traceability. However, we believe that progress can still be made and that the additional features of our traceability model, made available in a uniform and consistent way, including its structuring facilities and the tracing of information not thought of in advance, is a valid contribution to the practice of requirements traceability.

1.6 Requirements for a Software Tracing Model

The tracing model we describe in this thesis is motivated by the following observations:

- Requirements evolve throughout the lifetime of a system. They change in nature, from general to specific; they change in scope, by incorporating new features and deleting old ones; and they change in content and in form, to become more consistent, precise and clear. They are also connected to each other and to other artifacts involved in system development. Moreover those connections also evolve over time.

- Requirements are *situated*, in that they depend upon details of the particular situation or context in which they arise. Requirements are negotiated in that they result from discussions among interested parties, and the outcome of which depends on parties' interests, organizational position, technical background, etc. Moreover, the relations of requirements to one another also arise through negotiation and depend upon context. For example, to say that a requirement is *derived* from some document may have different meanings in different contexts.

- Requirements are an inextricable part of the development process, and tracing project artifacts forward and backward from requirements is useful throughout the lifecycle.

It is clear from the above points that we view requirements tracing as related to the development process as a whole and not restricted to a particular requirements phase. Therefore,

the requirements of our tracing model come both from the general concepts just discussed and from an analysis of software development practices.

One important part of a system development environment is the way the process is conducted: the steps taken from its inception to its deployment, the activities performed, the interactions between them, the techniques used, the management constraints, etc. Life cycle models have been used for management purposes to assess and control the development of software from its inception to its deployment and maintenance [85, 14].

In the following subsections we describe the requirements related to the process of software development by analysing some existing software life cycle models and some desirable characteristics not always possessed by them. We focus mainly on the similarities between the different models. This gives us the necessary information to meet our first requirement.

Requirement 7 *The tracing model should not be tied to any particular way of developing software.* □

1.6.1 Similarities in Life Cycle Models

Life cycle models have two distinct aspects. The technical side is dependent on the particular methods and techniques used to develop systems. These methods and techniques are usually divided in stages that resemble (part of) the phases of the life cycle. The other, and more important, aspect is the management of the process. As a management artifact, phases provide baselines and check points for the development. Usually validation procedures and completion criteria are specified for each phase. Also, the phases in a life cycle model are used to schedule and to plan the activities and personnel of a software development, e.g., programmers are assigned to the coding phase and probably not to the requirements phase.

The models used in our discussion are the waterfall model [126], together with its refinements [13, 33], the prototyping models [12, 29], specifically rapid prototyping [87], operational prototyping [28], and evolutionary prototyping [93], the spiral model [16], and some alternative models like the functional life cycle model [69] and the commonsense management model [149]. We do not describe these models here but some details are given when we discuss their common features.

Despite the diversity of software life cycle models and the new proposals that still come to light from time to time, in some way indicating that the existing ones are far from satisfactory, the models presented above are among the ones most commonly used in real software development projects. Also, they share the following common features:

- The existence of phases.

- The existence of a forward path to guide the development process.
- The existence of a backward path for reassessment of previous issues.
- The existence of relations between the artifacts used or produced in the context of a phase.

We discuss each of these features in turn, highlighting the aspects we use to draw the requirements for our traceability model.

1.6.1.1 Existence of Phases

Life cycle models are based, in one way or another, on the concept of 'phase'. This reflects the basic software engineering principle of 'separation of concerns'. The idea is to maintain an adequate level of abstraction with respect to the activities being performed and not to mix activities from distinct levels of abstraction. In this way, for example, while discussing requirements one does not consider design issues, and while discussing design one does not bring up implementation issues[2]. Some models do prescribe a mix of these activities for some phases. For example, in the prototyping phase, when eliciting requirements, the design and coding should be performed for the features being prototyped. However, these activities may be done in a quick and dirty way just to make the prototype run. Usually, after the prototyping phase the system should be designed and coded for production use.

Therefore, phases in a life cycle model group similar activities and determine, to some extent, the nature of these activities, the character of the people involved, the necessary resources, and the expected outcomes. For example, the activities in the requirements phase are related to the investigation and description of the desired system features, but not to the prescription of how these features should come into effect. Also, requirements engineers involved in this phase are expected to have some skills not necessarily possessed by professionals engaged in other phases, e.g., programmers.

The activities in each phase vary greatly depending on the method or technique being used to carry them out. The requirements phase, for example, may involve interviews, in site observations, document readings, etc., for the elicitation of requirements (for an account of several of these elicitation methods see [60]). They may also involve meetings to agree upon a set of requirements and the actual writing of the system requirements specification. Also, the products of a life cycle phase, and generally the artifacts used therein, depend on these activities and vary according to the particular method being used. Thus, an interview

[2]We present divergent views in Section 1.6.2

transcript is the product of interviewing people and a JSD diagram is a product of performing the activity of design through a JSD development method [75, 76].

Requirement 8 *It should be possible to associate trace units to phases of the development process.* □

1.6.1.2 Existence of a Forward Path

Phases in a life cycle are ordered in such a way that activities in one phase start after completion of activities of previous phases. In iterative models the completion of one phase may not be total. For example, some design may be done based on a small set of requirements which will have later additions. Nevertheless, there should be some requirements before design. Commonly, the activities in one phase are based on the outcome of some previous phase. For example, the design of a system should be based on its requirements, and the coding should be based on its detailed design.

The ordering is usually sequential. There is a sequence on (some) phases, even when they are run in parallel. For example, in Davis's Waterfall Model [33] the planning of a system test is based on the outcome of the requirements phase which is also the basis for the design phase. Therefore both the design and the planning phases may occur in parallel but only after the completion of the requirements phase. In other models where the outcomes of a given phase can be partitioned in cycles, as in the Spiral Model [16], or in steps, as in the Evolutionary Prototyping Model [93], there is still a sequence for each cycle or step. For example, the first evolutionary prototype should be based on the initial requirements for the system, the second prototype should be based on the requirements gathered from the use of the first prototype, and so on.

The ordering of life cycle phases provides a natural forward path for the development. The system development goes (or would go in an ideal situation) from a very abstract description to a concrete realization and the steps taken are guided by the phases of the life cycle model being followed. It also provides a natural forward path with respect to the artifacts used in each phase.

Requirement 9 *It should be possible to define a forward ordering on the phases of the development process. It should also be possible to define traces of trace units associated with one development phase on trace units associated with the subsequent phases.* □

1.6.1.3 Existence of a Backward Path

In all life cycle models there is a recognition of the need for reassessment of previous phases. This is an acceptance of the fact that in real projects things rarely happen in a straightforward

way. The backward links between phases are an attempt to ensure a consistent reworking of all that has to be reworked. If some aspect of one phase has to be reworked the natural step is to verify the related aspects in the previous phases. This procedure can go all the way back to the initial phase and determines a backward ordering on phases.

The backward ordering of phases is usually the inverse of the forward ordering; for each forward step there is a corresponding backward step between the phases involved. This backward path also applies to the artifacts used in each phase. For example, an artifact in a given phase relates backward to the artifacts in the previous phase on which it was based. Prototyping [87] and other iterative models replace some backward links by an interaction over some of their (prototyping) phases.

Requirement 10 *It should be possible to follow traces according to the backward ordering of phases.* □

1.6.1.4 Existence of Relations Between Artifacts

Our discussion of 'normal' forward and backward paths highlighted the existence of relations between phases and the existence of relations between artifacts across phases. These relations were described as a natural consequence of the fact that the products of a given phase are the basis for the artifacts produced in the next phase.

It is not only the end products of a given phase that may be related to each other. Although not explicitly formulated in the description of the life cycle models, the artifacts inside a given phase may relate to one another. In producing the outcomes of any phase several artifacts or resources may be used in intermediate steps for which the relations to other artifacts are not clear from the life cycle models, yet they still exist.

Some relations are determined by the development method being used while others may not be explicitly dictated by any particular method. As an example of the first case, consider the design of a system using the *Structured Analysis* [36] method. From the initial description of processes new ones are produced containing more detailed descriptions, and from these detailed processes further ones are produced which are still more detailed. The procedure continues up to the point at which the processes can be described by mini specifications which are the basis for the coding in the next phase. It is easy to see that the successive descriptions of processes relate to one another according to the decomposition criteria used in producing them. The existence of relations not explicitly dictated by a method, and even of relations involving things that lie outside the method, may be illustrated by the relation of a person to an interview transcript or by the relation of a requirement to an organization's department to indicate that the requirement is of interest to people in that department.

Requirement 11 *It should be possible to define traces that occur in the context of a phase, i.e, traces involving only trace units associated with the same development phase.* □

Requirement 12 *It should be possible to define traces involving trace units not associated with any phase of the development process.* □

1.6.2 Critical Visions of Life Cycle Models

Life cycle models are strongly criticized by those who view them as no more than artificial creations that do not adequately model the software development process. Most of those criticisms concern the actual *modus operandi* of software development. It is argued that the actual practices in software development consistently violate the precise and clean boundaries and orderly activity dictated by the life cycle models. We identify two streams of criticisms; the first one addresses the inapplicability of some assumptions behind the life cycle models and the second one points to the impossibility of modeling the actual practices of software development to the extent life cycle models try to do.

1.6.2.1 Technically Oriented Criticisms

One instance of the first line of criticism is the duality between requirements and design. The argument that they cannot be viewed as separate activities and that in fact requirements and design should evolve together through an interactive process is an old one [106]. More recently McDermid [94] states that "it is, in general, impractical to produce complete, consistent and implementable requirements specifications, and *know* that this has been achieved, without modeling the system to be produced at the architectural level". In his approach from the overall system objectives and a small set of requirements derived from them, one should produce the definition of the system architecture. This architecture definition is then used to discuss, analyse, and build the system. Particularly, there should be no elaborated requirements specification since it is the specification of the system architecture which is 'signed off' by the interested parties.

A more general criticism is made by Jackson [75] regarding the top down approach of software development. He argues that in practice it is not possible to carry out such an approach in the linear way proposed by most of the life cycle models and, therefore, it is not realistic to expect the completion of an abstract view of the system before details of its functions are addressed. Generally the criticisms of life cycle models have in common the argument towards a more flexible way to use them in order to comply with real software development practices.

Requirement 13 *It should be possible to group trace units that are associated to different development phases without redefining the development phases. It also should be possible to define traces involving trace units grouped in this way.* □

Requirement 14 *It should be possible to redefine the development phases and the ordering on them. It should also be possible to redefine the associations between trace units and phases or to define multiple association among them.* □

1.6.2.2 Socially Oriented Criticisms

The second line of criticism is based on empirical observations of actual work practices. A current stream of research in software engineering uses methods and results from sociology, mainly ethnography and ethnomethodology, to draw conclusions applicable to the work practices of developing software systems. Relevant for our discussion is the work of Suchman [133] regarding the situated nature of the activities necessary to carry out plans. In her work Suchman argues that the use of plans is influenced by the context in which they are carried out, by the available resources and constraints, and by the social practices shared by the people involved. Ultimately, plans are viewed as socially constructed artifacts, with their procedures negotiated in each occasion of use.

In the same line of argument Button and Sharrock [24] present an interesting case study showing that actual practitioners are bound to organizational policies more strongly than to technical ones. In their study they present reasons why prescribed steps of a method or technique are not always followed by the people using these methods or techniques. The reason is again the fact that context and social factors play an important, and often unforeseen, role in the execution of the prescribed activities.

It is interesting to observe that some issues raised by those who argue for more attention to social factors are similar to the views expressed by people criticizing life cycle models with more 'technical' arguments: significantly, flexibility is a primary need to make life cycle models compliant with the socially situated practices of software development. Among the factors inherent to a social critique are the need to explicitly use the social context and to account for the role of negotiation.

Requirement 15 *It should be possible to adapt the model to particular project environments.* □

Requirement 16 *It should be possible to retrieve trace information in different ways.* □

Requirement 17 *It should be possible to define traces and trace units representing things that lie outside the technical chore of software development.* □

Requirement 18 *It should be possible to register as trace units the actual artifacts used in trace production.* □

1.7 Overview of TOOR's Tracing Model

Our tracing model is based on objects and relations among them. The model has hypermedia features so that the objects can contain actual artifacts like source code, binary code, design diagrams, etc. The objects and relations are created in the context of modules which are used to structure the development. They can be browsed and traced in different ways through the use of a small tracing language, a hypermedia browsing tool, and a graphical interface.

Following a good principle of software engineering practice, we reviewed existing tools and approaches to learn from previous experience. This review provides evidence for the appropriateness of one important aspect of our tracing model: the use of relations to model traces. The languages, tools, models and approaches reviewed vary in purpose and adopt different solutions, most of them not primarily related to requirements traceability. Nevertheless, regarding the traceability aspect they all share a common point: relations are the basis for tracing. Relations may not be explicitly mentioned, but they are present:

- in the models, specifying the possible connections between model elements;
- in the languages, through specific language constructs used to relate one element to another;
- in the databases, through explicit links or through attributes of database objects holding references to other database objects; and
- in the extraction facilities which allow the extraction of elements related by attributes or keywords having the same value.

This view of using relations for tracing purposes is not new. It was stated before by Ramamoorthy et al. [116]: *Traceability refers to the ability of tracing from one entity to another based on given semantic relation definitions.*

Some of the models and approaches reviewed are discussed and compared to TOOR in Chapter 8. For now we just provide an overview of the features of our tracing model, highlighting the way the concepts and tracing aspects presented previously are incorporated into it.

1.7.1 Defining Traces

The trace units are viewed as objects organized in classes. The existence of a trace of one object on another is represented by a relation between the two objects. Relations are also defined as objects and axioms are used to specify properties of relations, to constrain the way objects may be related, and to specify composition of relations.

1.7.2 Producing Traces

Traces are produced as objects are related in the context of TOOR. The creation of objects and relations is through templates. For each class TOOR automatically generates a template containing a field for each attribute in the class definition. The templates are shown in a menu from which the user chooses the desired ones. The production of traces involves registering the appropriate objects and the relations between them. To register an object the user fills in an appropriate template. Objects' attributes can be associated with files stored in different media.

1.7.2.1 Off-line and On-Line Trace Production

The use of templates to register relations corresponds to the off-line trace production. For special kinds of documents it is possible to generate traces and trace units automatically. The on-line production of traces is accomplished in the following situations:

- There is an automatic generation of relation instances according to the axioms of each relation. For example, if a relation R is defined as transitive, then every time an object a is related to b and b is related to c through R, the object a is automatically related to c through R.

- The use of TOOR is most effective when applied to object-oriented software development, particularly using FOOPS [123, 129] as the specification and coding language. Automatic relations among specification texts and code are generated based on the module structure of the FOOPS programs. However, the model has general characteristics and the generation of automatic relations based on the structure of specification texts and programs can be incorporated into it for other languages and development methods.

- The objects and concepts used during the various phases of development should be those accessible from the module in which they are being used. For example, if a design module is specified to contain a class `CheckAccount`, the concept of `CheckAccount`

should be accessible for this design module. If it is accessible an automatic relation between the concept and the class definition is generated.

1.7.3 Extracting Trace Information

Deciding to trace something involves some motivation. In a system with hundreds (and often much more!) of interrelated objects, getting all the information available may not be useful. There may be so much information that the configurations before and after tracing are almost the same. Selective tracing is necessary in such cases. However, the criteria to be used in selecting information may not be so clear. A user can often get valuable insight by 'browsing around', guided by experience and background; browsing can be constrained to objects connected to a certain selected point. In other cases, the user may be able to make very precise selections. To support a variety of situations, TOOR provides the following three trace modes:

- **Selective Tracing** restricts tracing to certain selected patterns of objects and relations.
- **Interactive Tracing** allows interactive browsing through the objects related to a selected object. The choice of backward or forward tracing can be changed at any time. Moreover, a backtracking mechanism makes it possible to return to any previous point in the trace.
- **Non-guided Tracing** allows a user to go from one object to another at will, inspecting contents as desired.

1.7.4 Functional, Nonfunctional, and Environmental Tracing

Functional tracing is achieved by using relations between objects representing the functional aspects of the system. Nonfunctional tracing is achieved in the same way — by using relations between objects representing the nonfunctional aspects of the system. Nonfunctional tracing is also achieved by the use of relations among groups of objects and between objects and modules. Environmental tracing is achieved by using relations between objects representing the environmental aspects of the system.

1.7.5 Architecture of the System

TOOR, the tool to implement the model, uses a *project specification* written in FOOPS [123, 129] to set up the environment in which a project is carried out. FOOPS (for Functional and Object-Oriented Programming System) is a general purpose object-oriented language with

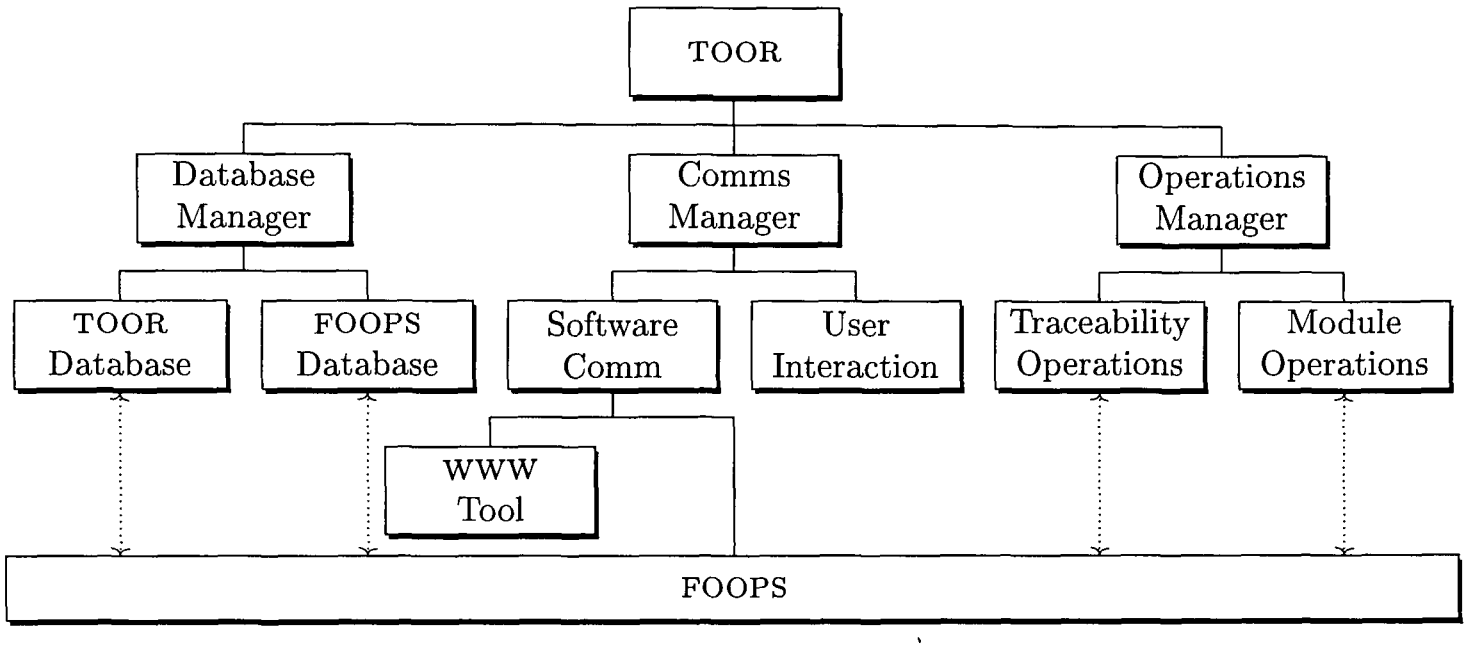


Figure 1.1: Structure of TOOR

specification capabilities. The project specification defines the trace units and traces as objects and relations, respectively. TOOR uses FOOPS-like modules to declare objects and to automatically create new links as a project evolves. The hypermedia features are provided through communication with specific hypermedia tools like MOSAIC. The system consists of the following components: an operations manager, a database manager, and a communications manager. The operations manager controls and executes all object and module operations, as well as the traceability mechanisms. The database manager controls access to the TOOR database, and uses the FOOPS database to get previously defined FOOPS modules. The communications manager controls communication with other systems, including FOOPS, design tools, and WWW tools. It is also responsible for controlling user interaction. These components are shown in Figure 1.1, where dotted lines indicate FOOPS connections to other components. FOOPS maintains and controls the TOOR database, and is used to evaluate most operations, objects, modules, and traces.

1.8 Preliminary Evaluation of the Solution

We adopt a model of software development based on the concept of module to organize the process and to give structure to development [49, 50, 51, 52]. This is an extension of the traditional view of software modules [103, 102, 104, 105]. The approach is called hyperprogramming and its principles were stated by Goguen in [53]. Hyperprogramming modules can hold not only software specifications, but also several related texts like requirements, diagrams, source code, binary code, etc. We advocate that they can also hold life cycle phases as will be explained in Chapter 3. In this way, software development has a uniform

characteristic throughout its evolution.

A useful way of viewing life cycle models is as retroactive snapshots as suggested by Goguen [55]. Plans are made to organize the work at a certain point in time, and later necessarily altered to cope with the inevitable situated practices of software development. The flexibility of modularization and the diversity of entities that can be declared and used inside modules make this view of software development adaptable to several existing life cycle models. This is an advantage since we are not bound to a certain way of developing systems and the adaptability to social practices is more secure. Therefore, we adopt the socially oriented criticisms of software life cycle models, but we also argue that the points highlighted in the previous discussion concerning the similarities among the existing life cycle models can be applied to our vision of the development process.

The tracing model we present in this thesis has specific features to address the concepts of trace definition, production and extraction. Its implementation, through TOOR, provides automatic production of some traces and flexible and powerful mechanisms for trace extraction. The TOOR hypermedia features make it adequate not only for functional and nonfunctional tracing but also for environmental tracing. The TOOR tracing model is intended to meet the requirements stated in Section 1.6. An assessment of the satisfaction of these requirements by TOOR is given in Chapter 8. However, the requirements are not precise enough. They cannot be used to objectively test a system. The normal procedure is then to refine each one and produce a set of precise, testable requirements which, in turn, will be requirements for a precise, specific model. In the development of our solution this refinement will be clear, reflected in the specific features of our model and represented in our design. This refinement towards specific TOOR requirements is difficult to question because it applies to a particular tool. Therefore, the assessment should be of an argumentative nature showing the reasonableness and applicability of our design decisions, which is demonstrated through examples of actual use of TOOR presented in the remaining chapters.

1.9 Thesis Contribution

This thesis contributes to the research on software requirements in the following ways:

- The definition of an abstract traceability model.

  The abstract features of our model meet our concept that traces, and therefore traceability structures and extraction procedures, cannot be thought of in advance for every possible situation. Thus, a model that can be instantiated to cope with particular situations and that has features allowing the

expression and retrieval of not previously thought trace information is a contribution to the practice of requirements traceability.

– The definition and formalization of a general traceability mechanism that may be used as part of other environments for requirements.

  Our tracing model uses a regular expression based language as a powerful mechanism to trace objects. The concept of a configuration state of objects and relations is precisely defined as are the associated concepts of a configuration grammar and configuration automaton. Also, the regular expression based language and the matching procedure involving a regular expression automaton and a configuration automaton are formally defined. Therefore, any environment for requirements that is capable of identifying objects and relations may use these concepts to dynamically build a configuration state to be used for regular expression based searches.

– The definition and development of an environment that may be used in requirements related research.

  The abstract features of our model and the existence of a formal way to define objects and relations and its user-definable and hyper-media features make TOOR suitable for use in requirements related research. It may be used to build and test models of requirements traceability. For example an IBIS-like [25] model may be constructed by defining appropriated classes for the concepts of *issue*, *position*, and *argument*, and for the relations between them. This use of TOOR is discussed to a great extent in Chapter 7. Also, TOOR may be used to investigate requirements capture in virtue of its ability to handle information in different media and to keep track of them.

- The presentation and validation of a general traceability model that take some account of the situated aspects of software development.

  This contribution relies on the fact that software development has indeed situated aspects that are important to account for. We strongly argue in favor of this but we also base our assumptions on a large number of research efforts sharing similar views [55, 57, 54, 60, 88, 133, 24, 63, 131, 73, 128, 99, 40, 41]. Of course, we do not claim that our traceability model accounts for all aspects

of situatedness which is indeed a notion subject to different interpretations. However, we do claim that TOOR contributes to the achievement of certain qualities attributed to situatedness under a particular interpretation [62]. The nature of these qualities and the discussion of how TOOR contributes to their achievement are presented in Section 9.3, where our model is evaluated.

- To a lesser extent the implementation of TOOR also contributes to the development of the *hyperprogramming* paradigm.

  Although TOOR does not intend to be an effective implementation of hyperpergramming, its use of FOOPS and its use of modules containing different kinds of information allows for a limited exploration of some concepts of the hyperprogramming paradigm.

1.10 Thesis Structure

The rest of this thesis is structured in the following way.

Chapter 2 introduces the concept of a project specification and explains the way a system administrator can set up an environment for a given project. A project specification formally defines the traces and trace units for a given project. This chapter gives a precise idea of what is involved in the definition and how the use of TOOR is determined by the contents of the specification. It also discusses how changes to a project environment can be made.

Chapter 3 discusses the use of TOOR modules and explains how the module structure of a project is used for requirements tracing. It also explains how FOOPS modules interact with TOOR modules. Some aspects of developing systems in an object-oriented way are discussed together with problems that can occur during the development. It is shown how tracing using TOOR's module structure can help in managing these problems. The module features of TOOR are illustrated by means of an extended example used throughout the chapter.

Chapter 4 presents details of the use of regular expression to trace requirements. It discusses the pattern matching procedure used to search objects and relations, and exemplifies the various forms of tracing with regular expressions. This chapter also explains how regular expression tracing interacts with other modes of tracing such as tracing using modules and browsing objects.

Chapter 5 discusses the use of hyper-media facilities to enhance traceability and presents the ways how objects and relations can be browsed in TOOR. It also gives details of how external files can be associated to objects.

Chapter 6 illustrates the use of TOOR to keep track of requirements evolution. It discusses the types of changes the objects can undergo during system development and explains how they can be managed in TOOR.

Chapter 7 shows how the specification features of TOOR, through a project specification, can be used to define domain models of objects and relations. It illustrates the discussion by applying TOOR to existing traceability models, therefore, stressing the point that TOOR can also be a useful research tool.

Chapter 8 summarizes TOOR's features and compares TOOR with other tools and approaches.

Chapter 9 discusses future directions for research and concludes our presentation with an assessment of our requirements for software tracing models and an evaluation of our model.

Appendix A contains the formal aspects of TOOR and some deferred proofs.

Chapter 2

Defining an Environment for Tracing

Traces occur naturally in software development. However, they have to be captured in some way if they are to be searched afterwards. The first step is to define the things that can leave traces, the things that can receive traces, and the nature of the traces that we want to capture. In a tracing model this amounts to defining the traces and trace units.

TOOR does not impose a fixed model of objects and relations among them. One important assumption of TOOR is that requirements are situated; they are negotiated and they result from discussions and interaction among the people involved. The project development itself is influenced by the environment in which it is carried out. The technical background of people, the administrative structure of the organization, the enforced policies, the political interests, and to sum up, the cultural environment give to each project specific characteristics. The understanding of what an object is, how they can relate, what meaning should be assigned to each relation, etc., is also dependent on the cultural environment of a project. Therefore, a tracing model should provide ways to define traces and trace units that are meaningful to each situation and that are in accordance with the expectations of people using the model. In TOOR the traces and trace units are defined through a project specification and may vary from project to project.

A project specification is a formal specification, written in FOOPS, defining the classes for the objects and relations that may be used when developing a project. Each class for the objects we want to trace, such as *People* and *Statement*, should be specified and each relation we want to use, such as *Assert* between *People* and *Statement* objects, should also be specified as a class in the project specification. TOOR uses the project specification to set up an environment according to the specified classes. For each class TOOR automatically generates a template containing a field for each attribute in the class definition. The templates are shown

in a menu from which the user chooses the desired ones. The project environment may be modified by changing the project specification and also by changing the way objects and relations are shown. The user can specify different template layouts by altering the ordering and size of the fields. The user can also suppress fields in a template and suppress templates in the template menu. The objects and relations can be viewed through a graphical interface and the user can modify the way objects and relations are shown. TOOR also implements tracing procedures according to what is specified for each class. The production of traces involves registering the appropriate objects and the relations between them. To register an object the user fills in an appropriate template. Object's attributes can be associated to files stored in different media.

This chapter describes how traces and trace units are defined and how this definition is used by TOOR. This is done in the following steps. First we discuss the use of relations to define and capture traces. Then we present the concept of a project specification, showing how it is used to set up an environment for tracing and how templates to create objects and relations are automatically built out of it. This is followed by a more detailed discussion of the structure of a project specification, in which we explain the constraints imposed by TOOR on the way a project specification should be written. We show how these constraints are embedded in a *standard* project specification which greatly simplifies the task of specifying a project environment. We proceed to give a detailed account of the different ways in which traces (relations) and trace units (objects) can be formally specified. Finally, we explain how changes to a project environment can be accommodated through corresponding changes in the project specification.

2.1 Relations as Traces

A trace is captured through relations among objects. We specify that there is a trace of one object on another by declaring some relation between the two objects. The nature of the trace — if it occurs by virtue of some action or because of some transformation upon the objects — is given by the particular relation used. In this way a trace defines a path or history, showing how some particular state came to be; each step of such a history is an instance of some particular class of relationship. To trace an object is then to follow its related objects according to some composition of relations. To define what possible traces may exist and may be captured in a given project amounts to defining what relations may be modelled by the traceability model underlying the project development.

In TOOR relations are modelled as objects having two attributes — source and target — to denote the objects being related. For each relation $R \subseteq A \times B$ there should be classes

representing A, B, and R itself. For each object of the class representing R, its source attribute should take values from objects of the class representing A and its target attribute should take values from objects of the class representing B. Therefore, trace units in TOOR are defined to be objects whose class represents the domain or codomain of some relation and traces are defined to be objects whose class represents relations.

The definition of traces should also specify the conditions under which two objects may be related. Most of the time this is given just by the relation domain and codomain. For example, take a relation *State* from *People* to *Requirement*. The only objects that may be related under *State* are those of classes *People* and *Requirement*: an object p of class *People* is related to an object r of class *Requirement* provided that p has stated r. However, there may be situations in which we want to impose further restrictions. For example, we may specify that only people from the technical support department can state database requirements. This may be accomplished either by restricting the relation domain and codomain or by specifying a predicate that should be satisfied by the objects involved.

To avoid confusion about classes intended to specify relations and classes intended to specify objects we use the following:

Definition 19 **Relation classes** are classes intended to specify relations and **relation instances** are objects of relation classes. **Object classes** are classes intended to specify objects that may take part in relations and **object instances** are objects of object classes. □

2.2 Project Specification

A project specification is a formal specification written in FOOPS [129, 123] containing the definitions of the objects that can be traced during the development of a project and the definitions of the relations intended to capture traces of one object on another. It is not a system specification, and does not concern the classes involved in specifying a system. Instead, it declares the classes of artifacts that can be used, e.g., `Document`, `VideoScene`, `Requirement`, `Specification`, `TestProcedure`, etc. Thus, it is relatively stable and can be used for a whole family of projects. However, a project specification can also be expected to evolve in the course of a project, since new classes of artifacts may have to be registered, and existing classes may have to be modified. Yet the classes of artifacts used in a project tend to stabilize over time, and overspecifying a project specification is not as dangerous as overspecifying the system itself; users need not use all classes in the project specification, and unused classes can be inhibited from appearing in TOOR menus. The point of the project specification is to allow each culture, organization, and team of practitioners to support

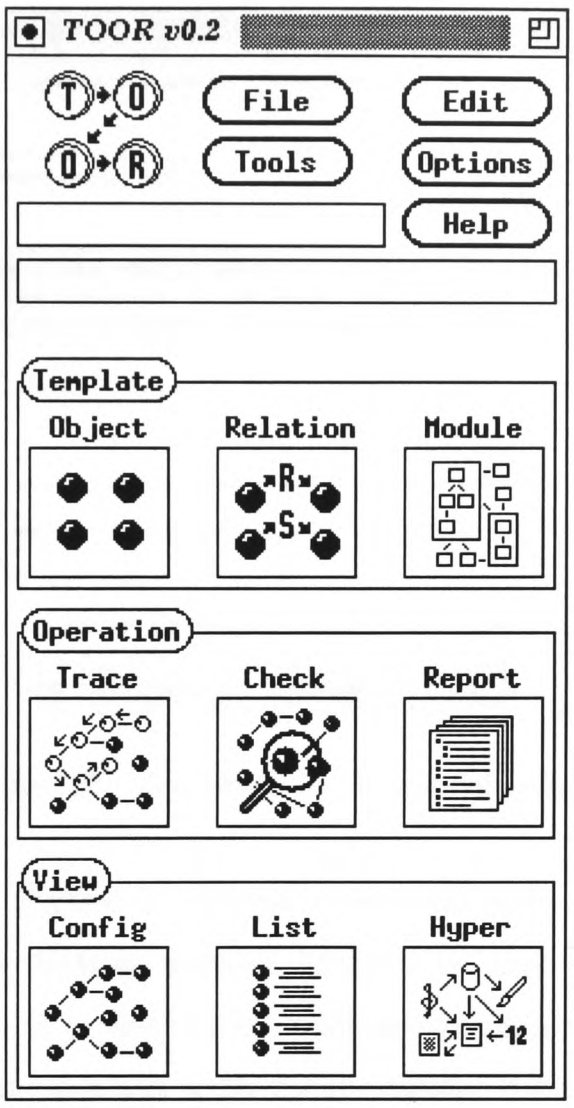


Figure 2.1: TOORMain Screen

its own values and practices, by defining their own classes of objects and relations, and interpreting them in their own context.

The project specification defines the possible traces and trace units for a project. It also defines an environment for the project, which consists of the menus and templates that TOOR supplies to register the actual objects used in the project development. The windows and processes through which the registered objects can be accessed and retrieved are also part of the project environment. TOOR constructs the project environment based on the information contained in the project specification. The following subsections describe how the environment can be used once the project specification has been defined.

2.2.1 Object and Relation Menus

There are two menus automatically created by TOOR whose entries are the identification of the classes declared in the project specification. The first is the *object menu* containing all object and the second is the *relation menu* containing all relation classes. When an entry from these menus is selected a template appears containing fields for the attributes and methods

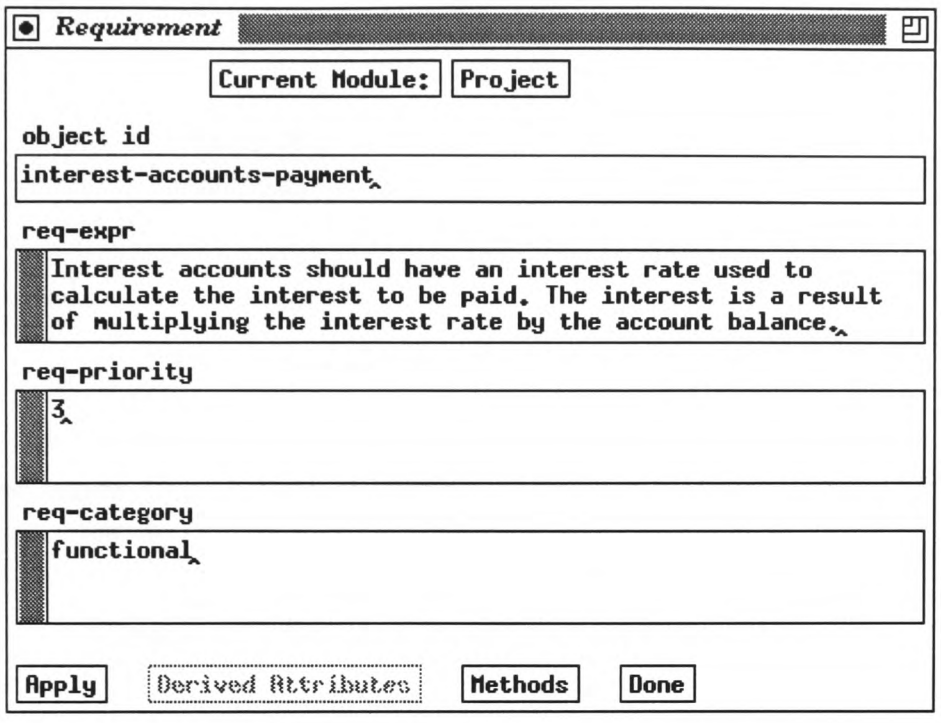


Figure 2.2: Template for Class `Requirement`

of the corresponding class. Figure 2.1 shows a screendump of TOOR. The object menu is activated by clicking on the button named 'Object' in the 'Template' part of the screen. Similarly, the relation menu is activated by clicking on the button named 'Relation'. The other buttons also activate menus but with fixed entries. We name menus after the name of the buttons used to activate them. Thus, the *module menu* is activated by clicking on the button labelled 'Module' and the *trace menu* by clicking on the 'Trace' button. The options in these and other menus are explained in later chapters, when appropriate.

2.2.2 Class Templates

For each class in a project specification TOOR generates a template containing one field for each attribute of the class. The template fields are initially blank for every attribute except for those attributes declared as having a default value. In this case the corresponding default value is shown. This saves typing when creating an object. Also, there are buttons to show all methods and derived attributes — whose values are computed by means of axioms — declared for the corresponding class. Figure 2.2 shows the template for a class `Requirement` containing the attributes `req-expr`, `req-priority`, with a default value of 3, and `req-category`. The `object id` field is common to all classes and the label 'Current Module' at the top of the template is a button used to set the module where the object is created. The module policy in TOOR is explained in Chapter 3.

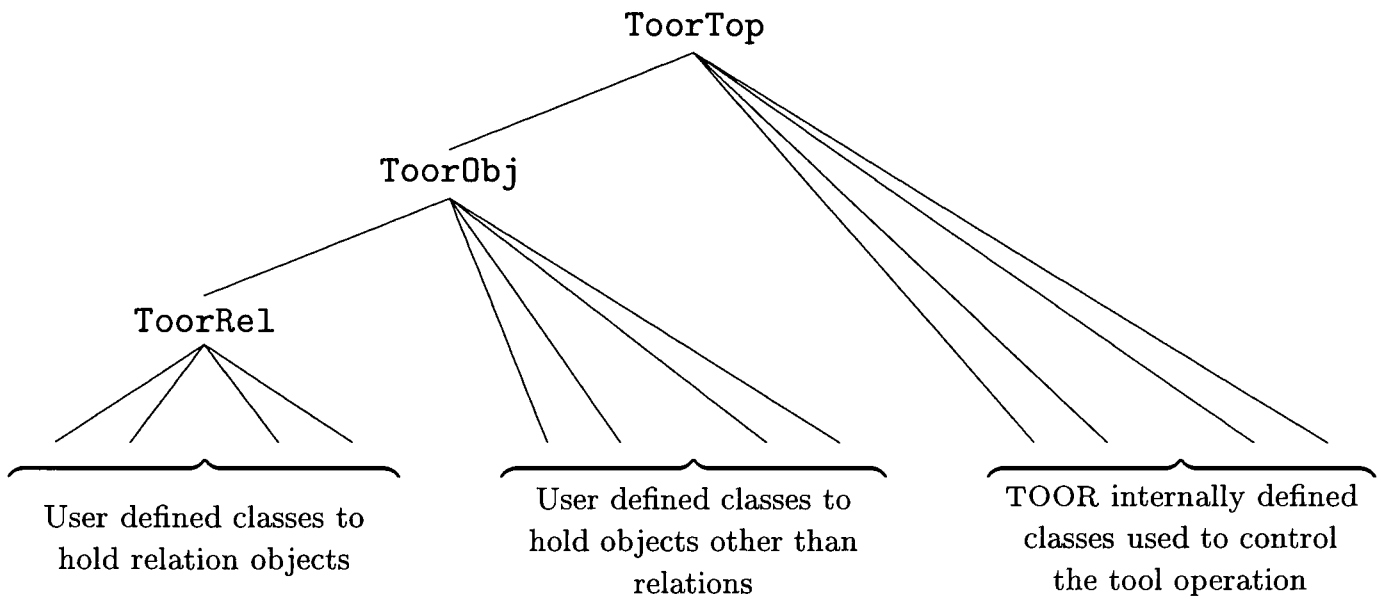


Figure 2.3: Project Specification Class Hierarchy

2.2.3 Creating Objects

Each time an entry from either the object menu or the relation menu is selected the corresponding template pops up. Default values for attributes are shown when the template pops up and do not need to be typed. Also, a method can be chosen to set or change the value for some attribute if there is such a method. An object is created by filling in the template fields with its attribute values and clicking on the `Apply` button.

2.2.4 Creating Relations

Relations are also created by filling in the corresponding templates with the identification for its source and target objects. If the relation has any attributes, they appear in the template and should be filled. The observations made for object classes about default values and methods also apply for relation classes.

2.3 Structure of a Project Specification

Every project specification should have three pre-defined classes called `ToorTop`, `ToorObj`, and `ToorRel`. Figure 2.3 shows the class hierarchy of a project specification. All classes defined by the user should be a subclass of either `ToorObj` or `ToorRel`. The class `ToorObj` is intended to be a superclass for all object classes, i.e., classes created to hold objects other than relations. The class `ToorRel` is intended to be a superclass of all relation classes, i.e., the classes created to hold relations — objects relating other objects. Since relations are also objects, the class `ToorRel` is a subclass of `ToorObj` which is a subclass of `ToorTop`. The class `ToorTop` also serves as a superclass for internally defined classes created to control the functioning of the model.

As in any FOOPS specification, a project specification also has a module structure. Classes are declared inside modules and are accessible from other modules through module importation operations. The context of a module consists of the module itself together with all other modules imported by it. A class is said to be in the context of a particular module if it is declared in that module or in any other module imported by it.

A project specification should have a module named `TOORFRONTEND` whose context contains all object and relation classes comprising the project environment, i.e., all classes declared to be used in a project. This is the only restriction about modules that should be satisfied when writing a project specification, although TOOR imposes other restrictions on the way a project specification should be written. However, there is a *standard* project specification with pre-defined modules satisfying all restrictions imposed by TOOR that can greatly simplify the specification of a project environment.

Figure 2.4 shows part of the standard module structure of a particular project specification. The pre-defined modules are underlined. The module `TOORTOP` declares the classes `ToorTop`, `ToorObj`, and `ToorRel`. The modules `OBJ-A`, `OBJ-B`, and `OBJ-C` are created by the user to declare classes for objects of type `A`, `B`, and `C`, respectively. Each one imports `TOORTOP` to declare its object class as a subclass of `ToorObj`. The module `TOORREL` is a parameterized module used as a generic pattern to declare a relation between objects of class `X` and `Y`, where `X` and `Y` are the parameters of the module. To declare a particular relation the generic module should be instantiated with the modules containing the classes to be related and the relation class should be renamed. For example, `RELATION-1` declares a relation between objects of classes `A` and `B`. It imports the generic module `TOORREL` after it has been instantiated with modules `OBJ-A` and `OBJ-B`. The module `RELATION-2` is created in a similar way. The module `TOORLINK` is a pre-defined module containing declarations to control the extraction of properties of relations such as its domain and codomain and its type — whether, for example, symmetric, transitive, or reflexive. This module is imported into the generic module used to declare relations. Finally, all relation modules are imported into the pre-defined module `TOORFRONTEND` which gives the context that will be used by TOOR.

A project environment is set up by submitting to TOOR a project specification containing the classes for objects and relations that will be used. All classes should be in the scope of the `TOORFRONTEND` module. This allows the user to specify which classes to consider just by declaring their modules in the context of `TOORFRONTEND`. Therefore one can have a very complete project specification and use only subsets of it. Alternatively the classes in a project specification may be removed from a project setup by making them not appear in TOOR menus.

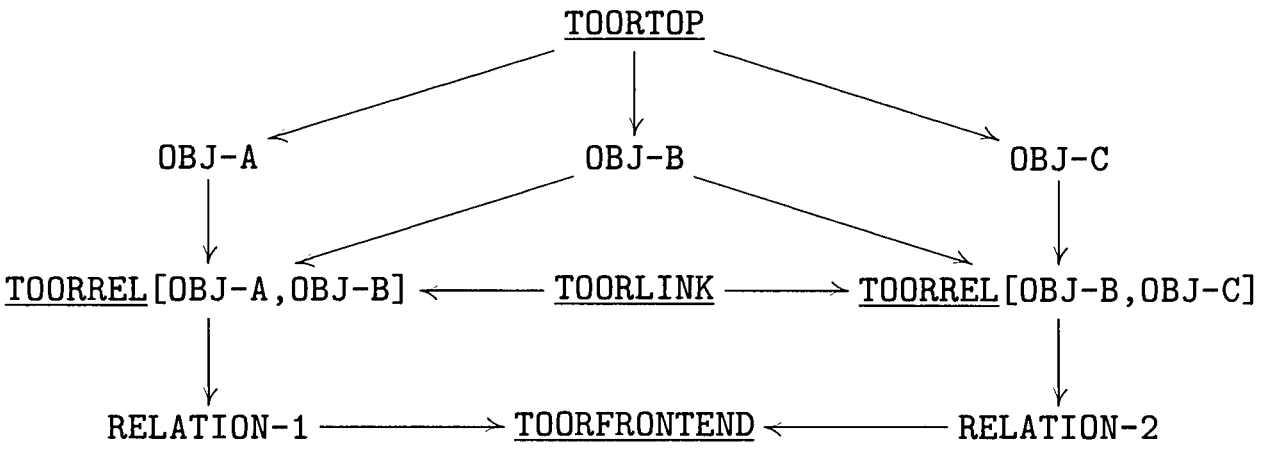


Figure 2.4: Standard Module Structure of a Project Specification

2.3.1 Project Specification Constraints

Apart from the obligation to have the class hierarchy described above and to use the module TOORFRONTEND as a context for all user defined classes, TOOR imposes a few other constraints on the way a project specification should be written. These constraints are necessary to the correct communication between TOOR and FOOPS, which has the following characteristics:

1) To build the environment of a project TOOR reads the project specification and retrieves all classes declared in the scope of `TOORFRONTEND`. A template is created for every retrieved class. For each class which is a subclass of `ToorObj` but not of `ToorRel`, TOOR creates an entry in the object menu. For those classes which are subclasses of `ToorRel`, TOOR creates an entry in the relation menu.

2) For every relation or object instance created or modified in a project TOOR automatically updates the attributes `t$creation` and `t$lastalter`. These attributes are not shown in templates but they are used internally for some bookkeeping procedures.

3) For every relation instance created in a project TOOR identifies the related objects by inspecting the contents of the attributes `t$source` and `t$target`.

4) Before relating any two objects `o1` and `o2` under a relation `r`, TOOR verifies if they can be related by means of the command `can-relate(r,o1,o2)`.

5) In its tracing procedures, TOOR verifies if two objects `o1` and `o2` are related under a relation `r` by means of the command `is-related(r,o1,o2)`.

The points above are translated into the following constraints that should be satisfied by every project specification.

Constraint 20 Every object and relation class should have attributes named `t$creation` and `t$lastalter`. They are defined to contain the date and time of its creation and modification. □

Constraint 21 Every relation class should have attributes named `t$source` and `t$target` taking objects as values. □

Constraint 22 The project specification should contain an operator `can-relate` with signature `can-relate : Id ToorObj ToorObj -> Bool`, where `Id` is the sort for identifiers in FOOPS, `Bool` is the sort for boolean values in FOOPS, and `ToorObj` is the class for TOOR objects. This operator takes the name of a relation as an identifier and two objects, and returns `true` if the objects can be related under the named relation and `false` otherwise. □

Constraint 23 The project specification should contain an operator `is-related` with the same signature as `can-relate`, i.e., `is-related : Id ToorObj ToorObj -> Bool`. This operator takes the name of a relation as an identifier and two objects, and returns `true` if the objects are related under the named relation and `false` otherwise. □

These constraints are already specified in a standard project specification which contains the precise definitions for the operators, attributes, and datatypes mentioned above. When writing a project specification, a user can simply use this standard project specification and add the classes for objects and relations that he wants to exist in the environment of a project. The following section gives examples that show how the standard project specification is used in defining a project environment.

2.4 Standard Project Specification

The use of the standard project specification saves time and facilitates the task of specifying a project because it already contains most of the auxiliary code needed for a correct setup. It is also quite general and gives the person who is specifying a project a lot of flexibility. For example, the operators `can-relate` and `is-related` are implemented in a way that takes into account additional user defined axioms.

The standard project specification is kept in a file `toor-std-prjspec.foo`. Figure 2.5 shows an extract of a project specification using it. The first declaration is the inclusion of the standard project specification (`in toor-std-prjspec`) which contains the definitions for

```
in toor-std-prjspec
omod PEOPLE is
  class People .
  extending TOORTOP .
  subclass People < ToorObj .
  protecting TEXT .  protecting DEPTO .
  at name : People -> Text .
  at department : People -> Deptname .
endo
omod DOCUMENT is
  class Document .
  extending  TOORTOP .  protecting TEXT .  protecting NAT .
  subclass Document < ToorObj .
  at content : Document -> Text .
  at type    : Document -> Nat [default: (1)] .
endo
omod STOREDDOC is
  class StoredDoc .
  extending DOCUMENT .  protecting FILE .
  subclass StoredDoc < Document .
  at content : Document -> T$File [redef] .
endo
omod REQUIREMENT is
  classes Requirement FuncReq NonFuncReq .
  extending  TOORTOP .  protecting TEXT .  protecting NAT .
  subclass FuncReq NonFuncReq < Requirement < ToorObj .
  at req-expr     : Requirement -> Text .
  at req-priority : Requirement -> Nat [default: (3)] .
  at req-category : NonFuncReq  -> Nat .
endo
omod EXTR-SUPP is
  extending RELATION[DOCUMENT,REQUIREMENT] * (class Relation to Extract) .
  extending RELATION[PEOPLE,REQUIREMENT] * (class Relation to Support) .
  var E : Extract . var S : Support .
  ax t$type(E) = ordinary .
  ax t$type(S) = ordinary .
endo
omod EXTRACT2 is
  extending RELATION[DOCUMENT,view to REQUIREMENT is
            class CTriv to FuncReq . endv] * (class Relation to Extract2) .
  var E : Extract .
  ax t$type(E) = ordinary .
endo
omod TOORFRONTEND is
  EXTR-SUPP + EXTRACT2 .
endo
```

Figure 2.5: Standard Form of a Project Specification

the basic classes `ToorRel`, `ToorObj`, and `ToorTop` as well as definitions for common datatypes like `Text` and `T$File`. The standard project specification also contains all code necessary to make a project specification comply with the constraints required by TOOR. The project specification of Figure 2.5 specifies classes for people (`People`), documents (`Document`), stored documents (`StoredDoc`), and requirements (`Requirement`) which are divided in functional (`FuncReq`) and nonfunctional (`NonFuncReq`) requirements. Except for requirements, each one of these classes are defined in its own module. An object module definition starts with the keyword `omod` and ends with `endo`. The FOOPS syntax used in this example is explained below, but for the moment we note that each class definition is very simple: basically one declares the name of the class (by means of the `class` declaration) and the position of the class in the class hierarchy (by means of the `subclass` declaration), together with the class attributes (`at` declaration) and axioms if necessary. Most of the modules that should be imported for a correct specification are either pre-defined modules of FOOPS (case of `NAT`) or are defined in the standard project specification (case of `TOORTOP`, `TEXT`, and `FILE`). Relations are defined by instantiating a generic module `RELATION` with modules containing the classes whose objects may be related. When defining a relation class, one just has to instantiate the module `RELATION`, rename the relation name (by means of the `*` declaration), and define an axiom (`ax`) specifying the type of the relation . The module `RELATION` is defined in the standard project specification. Finally, the module `TOORFRONTEND` is declared to contain all classes that will be used to set up the project environment.

When the specification of Figure 2.5 is submitted to TOOR, templates for every class in the context of `TOORFRONTEND` will be generated and the creation and manipulation of objects in TOOR will be bound to what is specified in it. For example, a relation `Support` may only be be created between an object of class `People` and an object of class `Requirement`. The following subsections give details of the specification of a project using the standard project specification.

2.4.1 Common Structure of Objects

Every project specification is based around the concept of a TOOR object. The pre-defined class `ToorObj` is a superclass for all other user defined classes. In the standard project specification the module `TOORTOP` below declares the common structure of all TOOR objects.

```
omod TOORTOP is
  classes    ToorTop ToorObj ToorRel .
  subclasses ToorRel < ToorObj < ToorTop .
  protecting TIMESTAMP .
  at t$creation  : ToorTop -> TimeStamp .
  at t$lastalter : ToorTop -> TimeStamp .
```

```
endo
```

It defines the classes `ToorTop`, `ToorObj`, and `ToorRel`; and the corresponding class hierarchy. The only attributes for these classes are `t$creation` and `t$lastalter`. The datatype `TimeStamp` is specified in the module `TIMESTAMP`. These attributes are used by TOOR to control versions and modification of objects. They are automatically maintained by the system and their values do not need to be given by the user. All other attributes should be specified by the user when declaring particular classes to be used in a project.

2.4.2 Specifying Objects

An object class is defined by simply declaring it as a subclass of `ToorObj` and by specifying all the desired attributes and axioms. The definition should be in a module which imports (by extending) `TOORTOP`. The use of the standard project specification makes defining object classes easier by providing a standard form for their definition: to specify a class `A` in a module `MOD-A` the user declares the class name, imports the module `TOORTOP`, declares `A` as a subclass of `ToorObj`, and declares all desired attributes and axioms for `A`.

Example 24 We write again, for ease of reference, the module `DOCUMENT` of the project specification shown in Figure 2.5. It declares a class `Document` containing two attributes: `content` and `type`.

```
omod DOCUMENT is
  class Document .
  extending  TOORTOP .
  protecting TEXT .
  protecting NAT .
  subclass Document < ToorObj .
  at content : Document -> Text .
  at type    : Document -> Nat [default: (1)] .
endo
```

The modules `TEXT` and `NAT` are imported because they contain the definition for the datatypes `Text` and `Nat`, respectively. The datatype `Text` is used to specify that the value for the attribute `content` should be a text. The data type `Nat` is used to specify that the value for the attribute `type` should be a natural number. The declaration `[default: (1)]` specifies a default value for this attribute. Most of the basic datatypes like `Text` for texts and `Nat` for natural numbers are already provided either by FOOPS or by the standard project specification. □

As in any FOOPS specification, it is possible to declare more than one class in a single module. This reduces the number of modules in a project specification.

Example 25 The module below, taken from the project specification of Figure 2.5, declares several classes to hold requirements objects.

```
omod REQUIREMENT is
  classes Requirement FuncReq NonFuncReq .
  extending  TOORTOP .
  subclass FuncReq NonFuncReq < Requirement < ToorObj .
  protecting TEXT .
  protecting NAT .
  at req-expr     : Requirement -> Text .
  at req-priority : Requirement -> Nat [default: (3)] .
  at req-category : NonFuncReq  -> Nat .
endo
```

The classes `FuncReq` and `NonFuncReq` for functional and non-functional requirements, respectively, are subclasses of `Requirement` which is a subclass of `ToorObj` as required by TOOR. The attributes `req-expr`, for requirement's expression, and `req-priority` are attributes of `Requirement` and also, by inheritance, of `FuncReq` and `NonFuncReq`. The class `NonFuncReq` has `req-category` as an additional attribute. □

Every object module in FOOPS has a principal class which is usually the first class declared in the module. The principal class for the module above is `Requirement`. The principal class is used by FOOPS to make automatic mappings when a module is used as an argument of parameterized modules. As we will see later, relations in a standard project specification are defined by means of parameterized modules. Therefore, declaring several classes in a single module is most appropriate if the other classes are subclasses of the principal one and only the principal class is used as domain or codomain of relations.

2.4.3 Using External Files to Specify Attribute Values

The tracing model of TOOR incorporates hypermedia features that allow the use of actual artifacts stored in the file system. In that way files containing design charts, program code, pictures, and video scenes, for example, may be used as values for object attributes. The attributes intended to be associated to actual files should have the sort `T$File` which is defined to accept file addresses as values. In the standard project specification this sort is defined in the module `FILE`.

Example 26 The following module is used in the project specification of Figure 2.5 to define a class of stored documents whose contents in TOOR are taken directly from the file in which the document is stored. The module declares a class `StoredDoc` as a subclass of `Document`, which is defined in the imported module `DOCUMENT` (see Example 24). The `content` attribute is redefined to get values of sort `T$File` and the other attributes are not modified .

```
omod STOREDDOC is
  class StoredDoc .
  extending DOCUMENT .
  protecting FILE .
  subclass StoredDoc < Document .
  at content : Document -> T$File [redef] .
endo
```

The template for this class looks exactly the same as the template for the class `Document`. The difference is that the value of `content` should be the address of an existing file. □

When inspecting objects with attributes associated to external files the user may access the file using the hypermedia features of TOOR. The user can *hear* its content if it is a sound file or *watch* video scenes if it is a video file. The user can also refer to the contents of an external file when tracing objects. These features are discussed at greater length in Chapter 5.

2.4.4 Restricting the Value of Object Attributes

A module can be as complex as necessary. One can declare inside a module, for example, data types and axioms to characterize the value of some attribute.

Example 27 The module below is a modification of the module presented in the Example 25. Instead of using natural numbers to specify requirement priority, it declares a sort `Pr` to be the type of values of attribute `req-priority`. This is done to restrict the priority value to 1, 2, or 3.

```
omod REQUIREMENT is
  classes Requirement FuncReq NonFuncReq .
  extending  TOORTOP .
  sort Pr .
  protecting TEXT .
  protecting NAT .
  subclass FuncReq NonFuncReq < Requirement < ToorObj .
  fns 1 2 3         : -> Pr .
  at  req-expr      : Requirement -> Text .
  at  req-priority : Requirement -> Pr [default: (3)] .
  at  req-category : NonFuncReq  -> Nat .
endo
```

The declaration `fns` specifies 1, 2, and 3 as constants of type `Pr`. As there is no code to accept other values, these are the only valid values accepted for `Pr`. □

2.4.5 Using Axioms to Specify Attribute Values and Operations

The examples we have shown contain no methods and no axioms. The methods to modify attribute values of objects are automatically generated by TOOR which provides a way to replace old values for new ones. The user modifies the state of objects in the same way he creates them: using the templates and typing the new values for their attributes. For instance, in the Example 27 a new expression can be given to a requirement at any time and also a new number can be assigned to requirement's priority or category. A new value for `category` may be any natural number and a new value for `priority` should be one of 1, 2, or 3. Unless a different behaviour is wanted there is no need to declare such methods.

To use explicit methods to change the state of an object it is necessary to specify the axioms governing the application of the method.

Example 28 Suppose that in the specification of requirements classes a method is desired to increment requirement priority based on its previous value. The module below declares a method `incr-prty` to change the value of the priority attribute.

```
omod REQUIREMENT is
  classes Requirement FuncReq NonFuncReq .
  extending  TOORTOP .
  sort Pr .
  protecting TEXT .
  protecting NAT .
  subclass FuncReq NonFuncReq < Requirement < ToorObj .
  fns 1 2 3          : -> Pr .
  at  req-expr       : Requirement -> Text .
  at  req-priority   : Requirement -> Pr [default: (3)] .
  at  req-category   : NonFuncReq  -> Nat .
  me  incr-prty      : Requirement -> Requirement .
  var R : Requirement .
  cax  req-priority(incr-prty(R)) = 2 if req-priority(R) == 1 .
  cax  req-priority(incr-prty(R)) = 3 if req-priority(R) == 2 or
                                         req-priority(R) == 3 .
endo
```

The method is specified using conditional axioms (`cax`). The axioms specify that the priority value should be incremented by one every time the method is applied but no increment should be made if the value is already 3. Of course, this is only to illustrate the use of methods and method axioms in a project specification. To use this module in a real project there should be at least a similar method to decrement priority. □

There are two forms of method axioms in FOOPS: the Direct Method Axiom (DMA) and the Indirect Method Axiom (IMA). The following discussion on their differences is sufficient

for our purposes, further details are given in [129, 123]. Basically, for direct method axioms, an axiom should be specified for each attribute modified by the application of the method (as exemplified for method `incr-prty` of the previous example); and, for indirect method axioms, the axioms are evaluated as rewrite-rules and there is no need to explicitly specify their effect on object's attributes. The last form of FOOPS axioms is very useful for general programming and system specification, but less useful in a project specification. Direct method axioms are usually enough because, in a project specification, one is interested in just specifying the classes making up a project environment.

In every template there is a button labelled 'Methods' which activates a pull down menu containing all methods defined for the associated class. If the method axiom is defined as DMA with no arguments other than the object identification then it is automatically applied by simply making the appropriate selection in the method menu. The new values for the attributes modified by the selected method replace the old ones. If, otherwise, the method contains other arguments or if it is defined using IMA axioms then the user has to type the expression for the method in a small window that appears when the method is selected in the menu.

The attributes that have fields in templates are *stored* attributes. They are defined as those attributes for which the user should give their values. However, axioms may also be used to specify how values for some attributes can be calculated from the values of other attributes. The attributes whose values are specified in this way are called *derived* attributes. They are not stored in the FOOPS database and, consequently, they do not appear as fields in templates; the user need not type in their values since they are computed by axioms.

Example 29 The class `TestProc` for the specification of test procedure objects contains the attributes `testprc` for the text describing the test procedure, `planneddate` for the planned date of the test, `actualdate` for the date when the test is actually performed, and `timestatus` to give the status of the test regarding its planned and actual dates.

```
omod TESTPROC is
  class TestProc .
  extending  TOORTOP .
  subclass TestProc < ToorObj .
  protecting DATE .
  protecting TEXT .
  protecting TESTPROCSTATUS .
  at testprc     : TestProc -> Text .
  at planneddate : TestProc -> Date .
  at actualdate  : TestProc -> Date [default: (nulldate)] .
  at timestatus  : TestProc -> TestProcStatus .
  var T : Testproc .
```

```
    ax timestatus(T) = done    if actualdate(T) =/= nulldate .
    ax timestatus(T) = late    if actualdate(T) == nulldate and
                                  planneddate(T) > nowdate .
    ax timestatus(T) = planned if actualdate(T) == nulldate and
                                  planneddate(T) <= nowdate .
endo
```

The datatype `TestProcStatus` and its values `done`, `late`, and `planned` are defined in `TESTPROCSTATUS`. The datatype `Date`, the value `nulldate`, and the function `nowdate` giving the current date are defined in `DATE`. The datatype `Text` is defined in `TEXT`. When a `TestProc` object is created the value of its `actualdate` attribute is set by default to `nulldate`. The attributes `testprc`, `actualdate` and `planneddate` are stored attributes. The attribute `timestatus` is a derived attribute. At any time the value of the `timestatus` attribute is computed by the axioms according to the values of `actualdate` and `planneddate`. □

Although not shown as fields in templates, the derived attributes of any class can be inspected by the user. Every template has a button labelled 'Derived Attributes' that, when clicked on, shows a list containing all derived attributes of the associated class. The user may see the value of a derived attribute by choosing the appropriate entry in the list of derived attributes. Also, the values of derived attributes may be used when tracing objects. This feature is discussed with details in Chapter 4.

2.4.6 Specifying Relations

Relations in TOOR are declared as classes containing at least the attributes `t$source` and `t$target` to hold the objects being related. Every relation class in TOOR should be a subclass of `ToorRel` and the attributes `t$source` and `t$target` should be object valued attributes taking their values from objects of the classes representing the domain and codomain of the relation.

The generic module below is part of the standard project specification. It is intended to be instantiated with the modules containing declarations for the classes representing the domain and codomain of the relation being defined.

```
omod RELATION[X :: CTRIV, Y :: CTRIV] is
 class Relation .
 extending TOORLINK .
 subclass Relation < ToorRel .
 at t$source : Relation -> CTriv.X .
 at t$target : Relation -> CTriv.Y .
 at t$type   : Relation -> RelType .
endo
```

The parameter module CTRIV is a theory specifying syntactic and semantic constraints actual modules should satisfy to be used as arguments. In this case CTRIV contains only a single declaration for a class CTriv. Therefore, any module declaring a class may be used as an argument for RELATION. The module TOORLINK contains the declarations necessary to compute the mathematical properties of relations such as image, counter-image, etc., according to the relation type defined in the attribute t$type.

The datatype RelType together with the values it can assume are also declared in the module TOORLINK. The t$type attribute is intended to be a derived attribute, i.e., to have its value defined by an axiom. The relation types ordinary, transitive, reflexive, symmetric, and antisymm are predefined in the standard project specification. A relation is of type ordinary if its only related objects are those directly linked, i.e., those related by the user. The other relation types have the mathematical meaning implied by their names. They are used by TOOR to compute indirect links. For example, if a relation R on Dom is defined as reflexive, then every time an object of class Dom is created this object is considered as related to itself by R.

To declare a relation class using the standard project specification it is necessary to import the module RELATION, to instantiate it with the modules containing the class declaration for the domain and codomain of the relation, to rename the relation name, and to declare an axiom to specify the relation type.

Example 30 The module below declares a relation Extract on objects of class Document to objects of class Requirement.

```
omod EXTRACT is
  extending RELATION[DOCUMENT,REQUIREMENT] * (class Relation to Extract) .
  var E : Extract .
  ax t$type(E) = ordinary .
endo
```

The symbol * is a rename operator. In this case it renames the class Relation to be Extract. The principal class of DOCUMENT is Document and the principal class of REQUIREMENT is Requirement. Therefore, the t$source attribute of Extract takes values from Document and its t$target attribute takes values from Requirement. The relation type is set to ordinary. □

Using a module as an argument for a parameterized module requires a view to map the elements in the parameter theory to elements of the argument. In the example above the map is automatically done by default. As the parameter theory CTRIV has a single declaration for

class CTriv, this class is mapped to the principal class of the module used as argument, i.e., it is mapped to Document for the first argument and to Requirement for the second argument.

The form used in Example 30 is the standard way of declaring relations using the standard project specification. It is used to specify relations from the principal class declared in the module used as first argument to the principal class declared in the module used as second argument. However, it is possible to specify different mappings by means of explicit *views* defining the classes that should be used for the domain and codomain.

Example 31 The module REQUIREMENT of Example 28 declares three different classes for requirements objects, the class Requirement being the principal class of the module. If we want to declare a relation from Document to objects of class FuncReq we need to explicitly specify, in the second formal argument of RELATION, the mapping from CTriv to FuncReq.

```
omod EXTRACT2 is
  extending RELATION[DOCUMENT,view to REQUIREMENT is
                              class CTriv to FuncReq . endv] *
                                        (class Relation to Extract) .
  var E : Extract .
  ax t$type(E) = ordinary .
endo
```

This relation is similar to the Extract relation defined previously. The difference is that now it specifies a relation from the class Document to the class FuncReq. The view from the parameter theory to the module REQUIREMENT starts with the keyword view and ends with endv. □

2.4.7 Specifying Relation Attributes

Relations themselves can have attributes just as any other object. All observations made for object classes about the use of methods and axioms to specify attribute values are also valid for relation classes.

Example 32 Suppose that the relation Extract from documents to requirements indicates that the requirement's expression is taken from some textual passages in the document. Also suppose that there is an automatic procedure to extract requirements from documents, creating the requirements objects and the relation instances of class Extract to relate the extracted requirements to the document from which they are extracted. In this case we may want to register for each relation instance the extraction mode used. For example, the extraction mode may be defined as automatic if the relation instance is created by the procedure described above, and as manual if it is created by the user. The following module

declares a relation called **Extract** as in Example 30 but specifying an additional attribute **extr-mode** to register the way a requirement is extracted.

```
omod EXTRACT is
  extending RELATION[DOCUMENT,REQUIREMENT] * (class Relation to Extract) .
  protecting DOCEXTRACTION .
  at  extr-mode : Extract -> ExtractMode .
  var E : Extract .
  ax t$type(E) = ordinary .
endo
```

Here we assume that the sort **ExtractMode** is declared in the module **DOCEXTRACTION** and includes, for instance, the values **automatic** and **manual**. □

Also, as in an object module, more than one relation can be specified in a single module. In this case each relation should have its own axioms specifying the relation type attribute.

Example 33 The module below declares the relations **Support** and **Extract** each one with its own axioms to specify the relation type.

```
omod EXTR-SUPP is
  extending RELATION[DOCUMENT,REQUIREMENT] * (class Relation to Extract) .
  extending RELATION[PEOPLE,REQUIREMENT] * (class Relation to Support) .
  var E : Extract . var S : Support .
  ax t$type(E) = ordinary .
  ax t$type(S) = ordinary .
endo
```

□

2.4.8 Specifying Relation Axioms

The basic constraint two objects should obey to be related under a particular relation is that the source object should be of the class specified for the relation domain and the target object should be of the class specified for the relation codomain. This constraint is automatically defined by the coarity of the **t$source** and **t$target** attributes, i.e., by the class of the objects that may be used as their values. A second common constraint is determined by the type of the relation. For example if a relation **R** is specified as antisymmetric and the objects **o1** and **o2** are already related under **R**, then it is not possible to relate them in the reverse order. In the standard project specification this sort of constraint is already considered as part of the definition of the operator **can-relate**. However, it is possible to further restrict the way two objects may be related. For a relation named **R**, this further constraint is given by a method **may-R** with signature **may-R : ToorObj ToorObj -> Bool**. This method should be specified by the user if necessary.

Example 34 The following module declares a relation called `State` on objects of class `People` to objects of class `DbReq` (for database requirements) which is considered to be declared in the module `REQUIREMENT`.

```
omod STATE is
  extending RELATION[PEOPLE,view to REQUIREMENT is
                            class CTriv to DbReq . endv] *
                                        (class Relation to State) .
  me may-State : People DbReq -> Bool .
  var S : State .  var P : People .  var R : DbReq .
  ax t$type(S) = ordinary .
  cax may-State(P,R) = department(P) == technical .
endo
```

The relation name is `State` and, therefore, the method used to verify if two objects may relate is named `may-State`. The constraint axiom for `may-State` specifies that only people from the technical department can state database (`DbReq`) requirements. □

TOOR uses the axiom `may-R`, for a relation `R`, when evaluating the operator `can-relate` to see if two objects can be related under `R`. The operator `can-relate` is declared in module `TOORLINK` and consists of two parts: a fixed part called `may-relate` and a variable part that is based on the existence of the method `may-R`. When evaluated for a relation `R`, the fixed part results in `true` or `false` depending on the type of relation and the variable part results in `true` if there is no axiom for `may-R` or it results in the value given by the existing axiom. Thus, for a relation `R` the operator `can-relate(R,o1,o2)` expands to

```
may-relate(R,o1,o2) and may-R(o1,o2).
```

The constraint axiom gives more flexibility in the way relations may be declared. In the example above the same effect would be achieved if there was a class to hold people from the technical department and the relation was specified on that class. However, the specification for an object class may already exist and sometimes it is easier to add an axiom than to redefine an object class specification. Also, the axiom can be specified temporarily, during a certain phase of the project, and later removed.

2.4.9 Specifying Indirect Links

In the standard project specification the operator `is-related` is declared in the module `TOORLINK`. This operator when applied to a relation identifier with two objects as arguments results in `true` if the objects are related and `false` otherwise. The evaluation of the method takes into account the relation properties given by the type of the relation. In this way two

objects may be considered related even if there is no direct link between them. For example, if a relation R is declared as transitive and the object o1 is related to object o2 under R and object o2 is related to object o3 under R, then the object o1 is considered as related to object o3 even if there is no link between them, i.e., even if the user does not explicitly relate them.

Another type of indirect link by which objects may be considered related even if there is no link between them is given by composition of relations. For example, one may want to declare that a requirement r is *Derived* from a document d if it is derived from a document d' and d' is part of d, i.e, considering $PartOf \subseteq Document \times Document$ and $Derive \subseteq Document \times Requirement$ one may want to specify

$$(\forall d, d' \in Document)(\forall r \in Requirement)$$
$$((d, r) \in Derive \text{ if } (d', r) \in Derive \text{ and } (d, d') \in PartOf).$$

To specify that objects are related under a relation R through composition of other relations it is necessary to explicitly declare a method is-R, with signature `is-R : ToorObj ToorObj -> Bool`, together with its corresponding axioms.

Example 35 The situation described above may be specified in the following way:

```
omod DERIVE-PARTOF is
  extending DERIVE .
  extending PARTOF .
  me is-Derive : Document Requirement -> Bool .
  var D : Document . var R : Requirement .
  cax  is-Derive(D,R) = member(image('Derive,D),R) or
                inter(image('PartOf,D),image-1('Derive,R) =/= empty .
endo
```

The relation PartOf on Document to Document is declared in the module PARTOF. The relation Derive on Document to Requirement is declared in the module DERIVE. The method is-Derive is specified to result in true if there is a common element in the image of D under PartOf and in the counter-image of R under Derive. The functions member, inter, image, and image-1 for membership and intersection of sets and image and counter-image of relations, respectively, are all defined in the module TOORLINK. □

Observe that the user does not need to worry about how the method is-R, for a relation R, is going to be applied by TOOR. In the standard project specification the operator is-related is defined to automatically verify if there is any method is-R declared by the user. This is done in a way similar to the one described for can-relate.

2.4.10 Automatic Relations

Some relations are automatically made by TOOR. The most evident of these relations is the subclass relationship. If a class is declared as a subclass of another one then the objects of both classes are automatically related. The subclass relationship does not need to be specified as a relation class in a project specification. Also, the user does not need to register instances of this relation. Relations based on the class hierarchy are automatically computed and their instances are a direct result of creating and deleting objects. Information about subclass relationships may be used to trace objects. For example, one can trace a requirement back to documents whose class is a subclass of `Confidential`.

Other types of automatically generated relations are those computed as a result of the system development itself. In an object-oriented approach to system development the classes used to specify a system should reflect concepts of the world being modelled. This is true also for the attributes and methods which should characterize the concepts and the operations being modelled. TOOR uses this fact to automatically generate relations between a FOOPS specification object and concept objects that are registered in a project. The details of this type of automatically generated relations are given in Chapter 3 when we discuss the use of TOOR for object-oriented system development.

2.5 Project Evolution

Once a project specification is defined the user may start a specific project by creating objects and relating them. Of course, the project specification may be expected to evolve. A new class may be considered necessary to better register the relations between objects or an existing class may be modified. For example, on reaching the design phase there may be a need to register technical manuals as a specific subclass of `Document` or to create a new class to hold entity-relationship diagrams. Also, after some time it may be noticed that some classes have attributes that are never used and removing these attributes would facilitate the registration of objects. Relations may also be perceived as unnecessary for a particular project, or not detailed enough. Also, there may be a need for new relations. All these changes are accomplished by corresponding changes in the project specification followed by a reload of the project to effect them. The discussion below considers how to modify some aspects of a project environment once a project is already under way.

2.5.1 Adding a new class

Adding a class that is not a superclass of any already defined class does not change the previously registered objects. When the project is reloaded an entry is added to the object or relation menu (according to the type of the class) and a template for the class is created.

Adding a class that is a superclass of some already defined class may change the previously registered objects. Reloading the project causes an entry to be added to the object or relation menu and a template for the added class to be created. All objects of any class that is a subclass of the added one are re-evaluated. They may inherit new methods and attributes, or there may be changes in their previously defined methods and attributes. If the previously defined classes inherit no methods or attributes then their templates and objects are not changed.

Example 36 Consider the module below as belonging to a project specification used for some project.

```
omod MANUAL is
  class Manual .
  extending TOORTOP . protecting TEXT .
  subclass Manual < ToorObj .
  at title   : Manual -> Text .
  at content : Manual -> Text .
endo
```

Consider also that the objects `M1` and `M2` are objects already registered as manuals when the project specification is modified to include the classes `TechBook` for technical books and `TechDoc` for technical documents. The class `TechDoc` is added just to provide a superclass for all technical documents. It is specified as a superclass of `TechBook` and `Manual`. However, it does not create any new attributes or methods and, therefore, the class `Manual` does not change. The class `TechBook` is a new class that is not a superclass of any previous class. The new modules in the project specification are:

```
omod TECHDOC is
  class TechDoc .
  protecting TOORTOP . protecting TEXT .
  subclass TechDoc < ToorObj .
  at title   : TechDoc -> Text .
  at content : TechDoc -> Text .
endo
omod MANUAL is
  class Manual .
  extending TECHDOC .
  subclass Manual < TechDoc .
```

```
endo
omod TECHBOOK is
  class TechBook .
  extending TECHDOC .
  subclass TechBook < TechDoc .
endo
```

Reloading the project with this new specification causes entries for `TechBook` and `TechDoc` to be added to the object menu and new templates for each class to be created. The new templates have fields for the `title` and `content` attributes. As `TechBook` is a new class that is not a superclass of any other class, there are no objects to be changed. Also, the state of the objects of class `Manual` is not affected by the inclusion of `TechDoc`. Therefore the objects `M1` and `M2` do not change and the template for `Manual` remains the same. Of course, the new class hierarchy is enforced and the manuals `M1` and `M2` become also technical documents. □

On the other hand, if the added class causes the existing ones to inherit a new attribute or method then the templates for the existing classes are modified to include a field for the new attribute. Also, the already registered objects are changed and get a default value for the new attribute.

Example 37 Consider the same situation of Example 36 but now the new class `TechDoc` contains an attribute `tech-area` to indicate the technical area to which a technical document is related. For simplicity assume that the technical area is identified by numbers. The new modules for `MANUAL` and `TECHBOOK` are the same as in the previous example and the module `TECHDOC` is given by

```
omod TECHDOC is
  class TechDoc .
  protecting TOORTOP . protecting TEXT .
  protecting NAT .
  subclass TechDoc < ToorObj .
  at title     : TechDoc -> Text .
  at tech-area : TechDoc -> Nat  [default: (0)].
  at content   : TechDoc -> Text .
endo
```

In this new situation the class `Manual` inherits the new attribute. Therefore, its template is changed to show a field for `tech-area` and the objects `M1` and `M2` incorporate the new attribute with value 0. □

2.5.2 Changing classes

A class may be changed to incorporate new methods and attributes, remove old ones, or modify the signature of its methods or attributes, i.e., the sorts of their values and arguments. When incorporating new attributes the previously registered objects of that class get a default value for them. Incorporating new methods causes no changes to the state of the already registered objects. In both cases the template for the changed class is modified either to show the new attributes or to add the new methods to its method menu. When deleting attributes or methods the already registered objects lose the deleted attributes and the template for the class is modified to suppress the corresponding attribute fields or the corresponding entries in the method menu. Changing an attribute or method signature may cause some objects to become incompatible with respect to the new signature. There may be two possible actions: the incompatible objects may be removed from the database or they may get default values according to the new attribute signature. One of these actions should be selected by the user when reloading a project.

Example 38 Consider the project specification described in Example 37. Consider also that two other manuals were registered: the manual `M3` with `tech-area` = 4 and the manual `M4` with `tech-area` = 2. The other manuals, `M1` and `M2`, remain with the default 0 for the value of the `tech-area` attribute. The project specification is modified again to restrict the value of the `tech-area` attribute to 0, 1, 2, or 3. This is done by modifying the attribute signature to specify that the only valid values are those of sort `TechArea`.

```
omod TECHDOC is
  class TechDoc .
  protecting TOORTOP . protecting TEXT .
  sort TechArea .
  subclass TechDoc < ToorObj .
  fns 0 1 2 3  : -> TechArea .
  at title     : TechDoc -> Text .
  at tech-area : TechDoc -> TechArea [default: (0)].
  at content   : TechDoc -> Text .
endo
```

Reloading the project causes no change to objects `M1`, `M2`, and `M4`. Their `tech-area` attribute values are still compatible with the new signature. However, the object `M3` has the value 4 for its `tech-area` attribute which is incompatible with the new signature. The object can be either deleted from the project database or modified to get the default value 0 for its `tech-area` attribute. □

2.5.3 Removing classes

Removing an existing class causes the objects of this class to be removed from the project database. Also the TOOR menus are modified to reflect the change and the template for the removed class is suppressed.

2.5.4 Changing relation axioms

The axioms specifying relational composition or relation constraints are used dynamically to verify relation properties or to compute indirect links among objects. They may be modified without the project having to be reloaded. One should only remember that relations registered under the old axioms will still be in effect. If the project is reloaded, the relation objects are re-evaluated and the relation instances incompatible with the new axioms are removed from the database.

Example 39 Consider the project specification described in Example 35. There, an axiom for `is-Derive` results in `true` if a document `D` is related to a requirement `R` through a composition of relations `Derive` and `PartOf`. The module `DERIVE-PARTOF` was created only to specify this relational composition.

If this module is removed from the project specification then the axiom is lost and documents and requirements can no longer be considered to be related through a composition of `Derive` and `PartOf`. As only indirect links between documents and requirements were created as a result of using this axiom, removing it causes no change to the TOOR database. Therefore, the project does not need to be reloaded. □

Example 40 Consider the project specification described in Example 34. There, a restriction on how people may be related to requirements is specified through an axiom for `may-State`. Using this axiom only people from the technical department may state a database requirement. Consider also that `Jon` from the technical department is directly related to requirement `R1` under the `State` relation. The module `STATE` is modified to further restrict the people who can state database requirements.

```
omod STATE is
  extending RELATION[PEOPLE,view to REQUIREMENT is
                            class CTriv to DbReq . endv] *
                                      (class Relation to State) .
  me may-State : People DbReq -> Bool .
  var S : State .  var P : People .  var R : DbReq .
  ax t$type(S) = ordinary .
  cax may-State(P,R) = true if department(P) == technical and
                                    area(P) == support .
```

```
endo
```

In this new version the axiom for `may-State` specifies that only people from the support area of the technical department can state database requirements. In this case we have to reload the project. Otherwise, considering that `Jon` is not from the support area, the relation `Support` between `Jon` and `R1` will be incompatible with the new axiom. □

2.5.5 General Modifications

The changes discussed above are the most common to a project specification but any modification can be made provided that the resulting specification is a valid FOOPS specification and, in the context of `TOORFRONTEND`, complies with the constraints imposed by TOOR.

Generally, modifying a project specification and reloading the project causes the TOOR menus and templates to be modified, to reflect the new context of `TOORFRONTEND`, and all objects to be evaluated against the new specification. Discrepancies are solved either by changing objects, using default values if necessary, or by removing them. In any case a complete report is produced to show all effected changes.

2.5.6 Customizing the Interface

A project environment can be modified without changes in the corresponding project specification. These are mostly changes in the way objects and relations are presented, but there also may be changes in the context of a project.

removing classes. A class may be inhibited from appearing in TOOR menus. Therefore no objects of that class may be created. This provides great flexibility in adapting an existing project specification to particular projects. A project specification can be very general and actual projects may use only subsets of it.

removing attributes. An attribute may be inhibited from appearing in the template for a given class. As the user will never type a value for the attribute, it should always have a default value. This should not be a problem because the user also will never see the attribute content. However, care should be taken to verify that other attributes or methods do not depend on its value.

changing attributes. The order of the attributes in a template may be changed. This may be used, for example, to place at the top attributes considered to be more important or to place at the bottom attributes that may be optional. Also, the size of the field for each attribute may be changed. They are initially wide enough

to hold one line of input in a window that may be scrolled. Widening attribute fields allows the user to see more of large contents like texts.

changing menu hierarchy. The object and relation menus reflect the class hierarchy of the project specification. In the object menu, for example, the direct subclasses of `ToorObj` are shown and the other classes appear in submenus that pop up when the user click on a superclass entry. This may be changed by placing at a higher level in the menu hierarchy a class that is intensively used.

2.6 Discussion

The use of a project specification provides much of the flexibility necessary for effective use of a software tracing tool. It takes some account of the situated practices of software development, in which the objects to be traced, the ways of tracing them, and even the meaning attributed to each object and relation are likely to vary from project to project, depending on the cultural background of the people involved.

The project specification not only allows the specification of different classes for different projects, but it also promotes adaptation of tracing procedures to the changing nature of software development. The use of axioms to define relations among objects makes it possible to trace them according to links not previously envisaged, and even according to links not directly made by the user; axioms may be added and removed at any point in time to try different ways of viewing related objects or to cope with changing perceptions of how objects should be considered related to each other.

The project specification, being a formal FOOPS specification, allows the automatic verification of objects' features through the use of the FOOPS system to evaluate the objects registered in a project. This is used to automatically make the necessary changes to the project database when a project specification is changed in the middle of a project. This feature is also used to trace objects according to the result of evaluating some of their attributes. In this way one can trace, for instance, a document forward to objects of class `Requirement` whose `status` attribute is `late`. This is possible even if the value of the attribute is dynamically computed by means of axioms. This feature is fully explained in Chapter 4.

Chapter 3

Using Modules for Requirements Tracing

The use of modules to structure programs and systems has many advantages in terms of reliability, reuse, and efficiency, but requires adequate tool support for their manipulation. From the first proposals to systematically address module issues [103, 102] to more recent developments [52, 53, 145] much has been done to promote their use and to make precise what is involved in it. Parnas [103, 102] established the foundations of modules, introducing concepts and techniques for the specification of program modules. Burstall and Goguen [22, 23] were the first to specify precisely a semantics for specification modules. They also were the first to give a calculus of modules [59]. Other works have appeared addressing similar or related issues [10, 86]. Modularization principles apply not only to programs and specifications but also to systems [52, 53, 56, 145], yet 'specification' remains a strong concern of these extensions. This naturally directs research on modules to specification issues. For example, in applying OBJ-like modules to relational (constraint logic) programming [37] the central question is how solutions of a query q defined in a module P can be translated into solutions in a module P' importing P. These specification (and, therefore, semantic) issues are fundamental for a consistent use of modules. Nevertheless, here, our interest lies in a different aspect. We want to stress the point that

modules are a valuable aid for tracing.

Introductory Example 41 Consider a situation where an error related to an overflow of a bounded buffer is found in a certain program. The first step in fixing it is to understand the nature of the error (it may be caused by a simple mistake like incrementing the number of elements by two as each element is added to the buffer). This may be done by looking at the specification of the program and the code implementing it. If no discrepancy is found we proceed a step further to look at how the specification was developed. In this example the

buffer is bounded by N and its elements are taken from a file. Therefore, the specification of the file and the specification where N is defined are reasonable places to be investigated next. We may proceed in this fashion back to requirements, to requirements sources, or even to people who are able to solve any inconsistency found in the process. Of course, the order of investigation may vary. The point here is that looking for related information has sometimes a stepwise component by which the next step is determined by the information received in the previous steps. If a system is developed in a modular way the determination of the next step is facilitated by the module structure used in its development. When an error is found in a module A, the first step would be to look for information defined in A and then to look for information defined in modules imported by A and so on, because those contain the only information A can manipulate. □

Information hiding is the basic concept of module use. It may be understood as the process of hiding information in a module so that access to it is only possible through well defined interfaces. This greatly enhances reuse and has beneficial effects on the development and maintenance of systems. Provided that the public features of a module are not changed, one can modify information and data in a module and be sure that no other part of the system is going to be affected. We argue that this mechanism is also beneficial to trace information. Modular structure facilitates the tracing of information by providing valuable hints as to

1) the start of tracing (in which module is a given datum defined),

2) the objects to be traced (what is defined in a module),

3) the tracing space (what are the module connections), and

4) the direction of tracing (what are the next modules to trace in).

TOOR exploits all these facilities. TOOR modules are used to structure a project and the tracing mechanisms of TOOR can use the project structure to order searches or to provide boundaries for searching. More importantly, TOOR automatically generates trace information based on the use of FOOPS modules to specify (and code) the system developed in a project. TOOR modules cannot be considered an extension of FOOPS modules. Nevertheless, there is a well defined connection between them that allows TOOR to use FOOPS modules as a source for generating trace information.

A natural extension of FOOPS modules leads to the hyperprogramming paradigm [53] where modules, together with other kinds of texts, are organized in *clusters* around a central specification. TOOR modules also group FOOPS modules as well as other kinds of information,

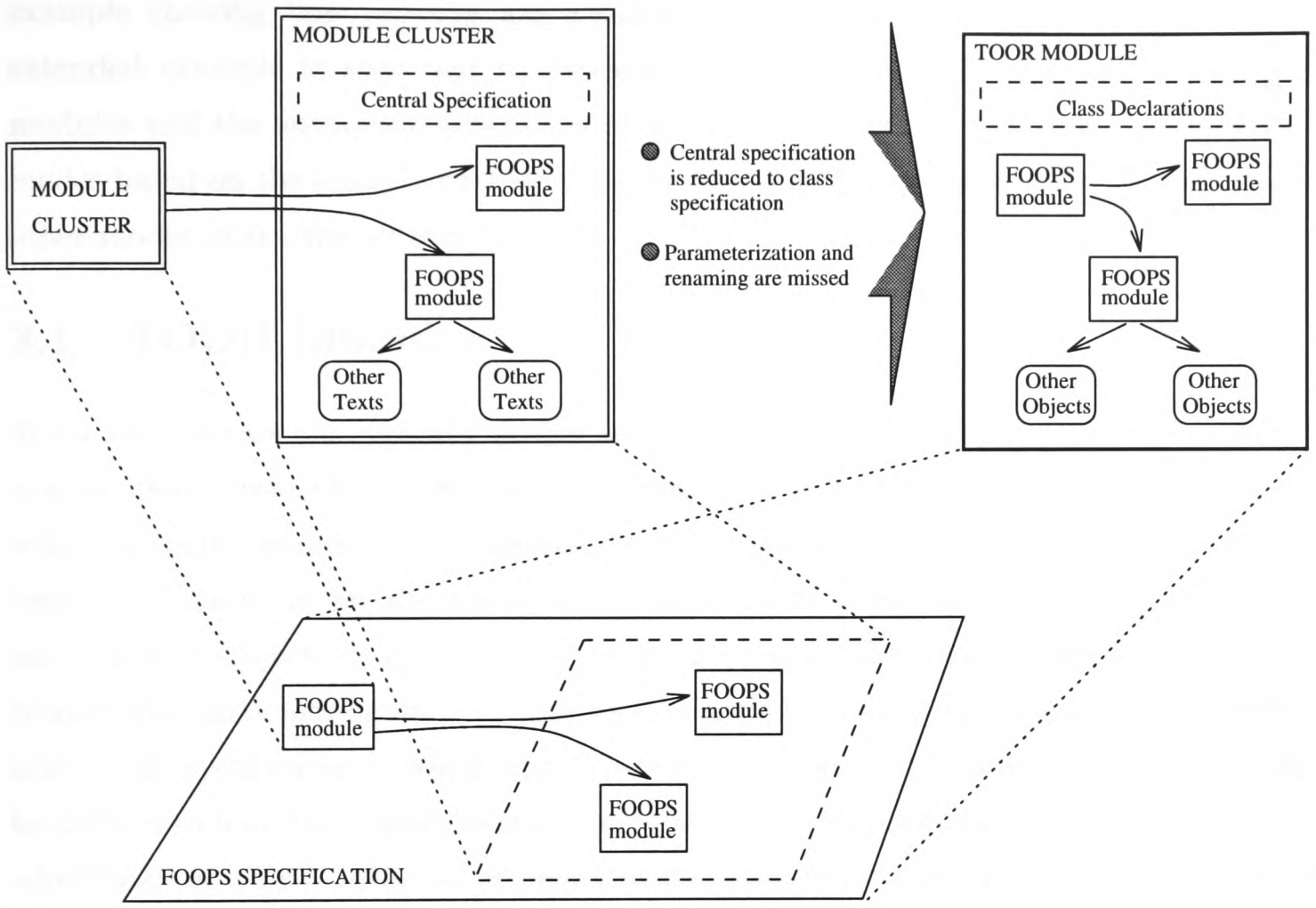


Figure 3.1: Module Clusters and TOOR Modules

but view them as objects. Figure 3.1 illustrates this point. The module cluster concept extends FOOPS modules by applying the same structuring and semantic features to other kinds of texts. On the other hand, a TOOR module just provides a scope for objects, and FOOPS modules are viewed in TOOR as a special kind of objects that are automatically related to each other according to their dependencies. The central specification of a cluster is reduced in a TOOR module to the specification of the classes of objects that may be created and used in it. In Particular, TOOR has its own policy for module importation and does not extend FOOPS module features like parameterization and renaming. TOOR modules are simpler than module clusters. However, they are powerful and flexible enough for tracing purposes. Furthermore, the missing features of FOOPS modules are missed only at a project level. They are present in TOOR, at a specification and coding level, through the very use of FOOPS.

This chapter explains in detail the use of TOOR modules for tracing objects in the following way. First we present an overview of TOOR modules, explaining their intended use, how objects are registered in them, and how they are interactively created by the user. Then we give details, alongside the module syntax, of class declaration and module importation. We also give details of how objects are registered in modules. This is followed by an extended

example showing how modules are actually used to structure system development. The extended example is also used to demonstrate the connection between TOOR and FOOPS modules and the automatic generation of traces. We proceed by showing two specific trace modes based on the module structure of a project and the use of modules in connection with other modes of tracing objects in TOOR.

3.1 TOOR Modules

TOOR modules are conceptually simple yet providing very general structuring facilities. They may be used to resemble system life-cycle phases but a project may also be carried out entirely within a single module. TOOR modules may be used for experimentation, e.g., to create a context to carry out an alternative development path where the objects created therein do not mix with objects in the mainstream development. They may be used to group objects related to a particular artifact, or they may be used to group related objects regardless of the system life-cycle phase to which the objects are associated. For example, a module `RSD` may be defined to hold the requirements taking part in the Requirements Specification Document which may be only a subset of all requirements objects created during the elicitation phase.

TOOR modules are used to structure projects by providing hierarchical scopes for objects used in a project development. They are not intended to structure the system being developed: that is the function of FOOPS modules. The main purpose of TOOR modules is to establish a space for object creation and a policy for object use. Objects in TOOR are registered inside modules and may be used inside the module where they are registered or in some other module that imports it. TOOR modules allow the user to structure the project development by defining scopes for object creation and use.

The signature of a TOOR module determines the classes of objects that may be created and used in it. These two aspects, object creation and object use, are clearly distinguished in TOOR modules. The creation of objects of a given class `C` may be allowed in a module `M` but not in a module `M'` even if `M'` imports `M`. Moreover, not only the creation of objects but also the use of objects created in `M` may be inhibited in a module `M'` importing `M`.

Example 42 Consider a module `REQS` whose signature allows the creation and use of objects of classes `Requirement` and `Derive`. This module is used to register requirements and requirements derivation, i.e., to create objects of class `Derive` relating objects of class `Requirement`. Now, consider a module `SPECS` whose signature contains the class names `Spec` and `Specify` for registering specifications and relations between a requirement and a specification, respectively. `SPECS` imports `REQS` in order to use the requirements already registered

in REQS. However, the intention is not to create in SPECS all objects that may be created in REQS. For example, one may want to restrict the creation of objects of class Derive to module REQS. This restriction could be extended to objects of class Requirement and, thereby, in SPECS one can only create objects of classes Spec and Specify. Objects already registered in REQS may still be used in SPECS. However, restrictions may also apply to object use. For example, objects of class Derive may have their use prohibited in SPECS meaning that they cannot be seen from this module and, therefore, they cannot be used in SPECS as source or target of relations. □

The signature of a TOOR module declares the classes of objects that may be registered in it. Any object registered in a module may also be used in that module, e.g., as source or target of a relation. We should remember that the specification of the classes allowed in a project is given by the project specification. Therefore, TOOR modules do not specify these classes, the modules just define where objects of a given class may be created and used.

A TOOR module B may import a module A preserving the class declarations of A. This means that the classes declared in A become part of B's signature and any object that may be registered in A may also be registered in B. Also, objects already registered in A may be used in B. This policy for object use is dynamically maintained: every time a new object is registered in A it becomes available for use in B. Alternatively, a module B may import a module A restricting the class declarations of A. This mean that B can only use objects of classes declared in A; they cannot be registered in B. The following sections show how these two kinds of importation are used to define scopes and policies for object creation and use.

TOOR modules are not specification modules. They are neither used to specify a project nor used to specify systems developed in projects. The former specification is given by the project specification and the latter is given by the specification texts written to this end which are viewed by TOOR as just another type of object. Nevertheless, a TOOR module does have a specification content given indirectly by the specification content of the objects registered in it. For example, a requirement specifies a system (partially and at a high level) by expressing some of its desired features. Therefore, a TOOR module containing requirement objects may be viewed as specifying the system through the expression of its requirements. The same is true about other objects that may have a specification content, e.g., FOOPS module objects.

TOOR is designed to be used in close connection with FOOPS. Particularly, software systems are intended to be specified using FOOPS. There are specific facilities to create and use FOOPS objects and to automatically maintain FOOPS module dependencies to other objects. There is even an option to 'extract' the FOOPS specification associated to a given TOOR module. That is, given a TOOR module M, the FOOPS specification associated to M consists

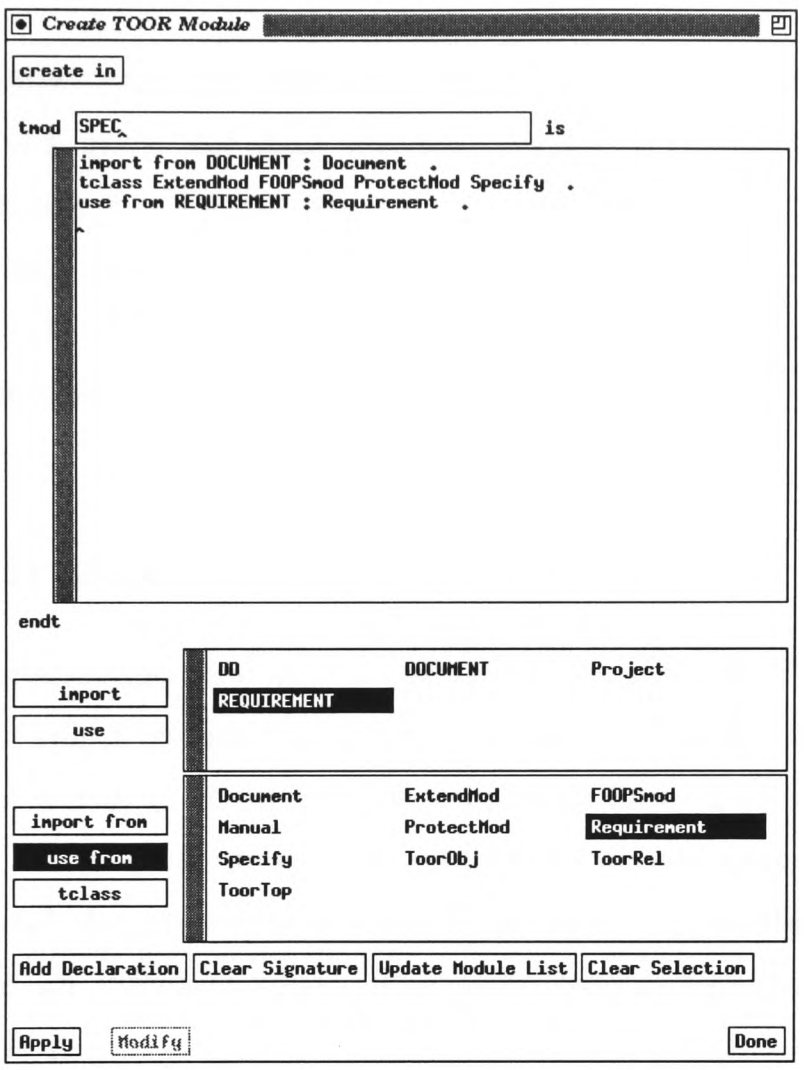


Figure 3.2: Creating Modules in TOOR

of all FOOPS objects registered in M. This close connection with FOOPS gives a powerful and complementary way of structuring systems: TOOR modules at a project level are used to organize the project development and FOOPS modules, at a specification and coding level, are used to organize the software development.

3.1.1 Interactive Creation and Use of Modules

TOOR modules are not intended to be written by the user. Modules and objects are created and used by means of menus and templates. The textual form of TOOR modules is just for internal use and is automatically maintained by TOOR as a result of interactions with the user.

To create a module the user selects the option *New Module* in the module menu of the TOOR main window. The template of Figure 3.2 pops up for the user to fill in the module characteristics. In fact, except for the name of the module, no information needs to be typed. The entire module signature may be built by clicking on, and selecting information from, appropriate buttons and lists of module and class names.

All fields may be clicked on to show a list containing the available modules and classes

that may be used.

The user may change a module signature at any time. Selecting a module to change its signature makes TOOR pop up the same template shown in Figure 3.2 containing the information presently associated with the selected module. The change of a module's signature is subject to consistency checks. For example, the signature of a TOOR module contains class declarations for the objects that may be registered in it. Then, if an object of class `C` is registered in a module, the signature of that module cannot be changed to remove the declaration for class `C`.

3.2 Module Syntax

TOOR modules are declared and used in a completely automatic way. The user uses menus and templates to create or select the module where objects and relation instances are registered. Thus, the user never writes a TOOR module and, therefore, does not need to know their syntax. Nevertheless, we present the syntax of TOOR modules because it helps the discussion of module features and the presentation of our examples. The discussion here, although rigourous, is kept on an informal tone. The more formal aspects of TOOR modules are presented in Appendix A.

A TOOR module is delimited by the keywords **tmod** and **endt**. The following is a declaration of a module named `REQ-ELICT`:

```
tmod REQ-ELICIT is
   body of the module
endt
```

The name of the module comes before the keyword **is** which introduces the body of the module. There are two types of declarations that may appear in the body of a TOOR module: declarations for the classes that make up the module signature and declarations for the objects that are actually registered in the module.

3.2.1 Declaring Classes

The signature of a TOOR module consists of the classes whose objects may be registered and used in it. When an object is created in TOOR it is always registered in some module. At any time TOOR has a current module which, unless otherwise stated, is the module in which the object is registered. For an object to be registered in a module `A` its class should be declared for object creation in the signature of `A`. We say that a class is declared for object creation in a module `A` if objects of this class may be registered in `A`. In the same way, a class is said to be declared for object use in a module `A` if objects of this class may be used in `A`. A class

is declared for object creation by means of a **tclass** declaration. The module REQ-ELICIT below declares Requirement and Document in this way.

```
tmod REQ-ELICIT is
  tclass Requirement .
  tclass Document .
endt
```

This type of declaration consists of the reserved word tclass followed by one or more class names. Any object of class Requirement or Document may be registered in REQ-ELICIT. The registration procedure is in fact a by product of the creation of objects in TOOR. Suppose that REQ-ELICIT above is the current module, then any object of class Requirement or Document created by the user becomes automatically registered in REQ-ELICIT. If the user tries to create an object of any other class, say Extract, the system issues a message saying that the object cannot be created because it cannot be registered in the current module. The user should either change the current module or alter its signature, declaring the class Extract for object creation.

A class may be declared as private to a given module by appending the keyword [private] to its name in a tclass declaration. The module REQ-ELICIT is rewritten to exemplify this feature.

```
tmod REQ-ELICIT is
  tclasses Requirement Document .
  tclass   Extract [private]  .
endt
```

The module above also illustrates the use of tclasses as a synonym for tclass. If REQ-ELICIT, in its last form, is imported into another module, the classes Requirement and Document become part of the signature of the importing module but not the class Extract.

3.2.2 Importing Modules

A module may import another by means of an **import** or **use** declaration. The **import** declaration preserves the way classes are declared in the imported module, i.e., if a class Requirement is declared for object creation in a module REQ-ELICIT which is imported (by import) into a module DESIGN, then objects of class Requirement may also be registered in DESIGN. The **use** declaration means that all imported classes are declared, in the importing module, for object use only; that is, in the situation described above, with REQ-ELICIT being imported by means of a use declaration, objects of class Requirement registered in REQ-ELICIT may be used, but no Requirement object may be registered in DESIGN. The

use of an object in a module is characterized, for example, by referring to it when creating a relation instance. In this way we may have a relation instance registered in a module, say DESIGN, which relates objects registered in a different module, say REQ-ELICIT. Both declarations are followed by one or more module names as in

```
import REQ-ELICIT SYSTEM-TEST .
```

The following examples illustrate how they are used.

Example 43 The module REQ-SPEC1 below imports REQ-ELICIT in such a way that the classes Requirement and Document are declared for object creation. Therefore, objects of these classes may be registered in REQ-SPEC1 and those registered in REQ-ELICIT may be used in REQ-SPEC1. Additionally, this module explicitly declares the classes Spec and Specify intended, respectively, for specification objects and for relation objects involving a requirement and a specification. Relation instances of Specify may be created involving objects from the same module (e.g., both registered in REQ-SPEC1) or objects from different modules (e.g., a requirement registered in REQ-ELICIT and a specification registered in REQ-SPEC1).

```
tmod REQ-ELICIT is
  tclasses Requirement Document .
  tclasses Extract [private] Derive [private] .
endt

tmod REQ-SPEC1 is
  import REQ-ELICIT .
  tclasses Spec Specify .
endt
```

The module REQ-SPEC2 is similar to REQ-SPEC1, differing only in the importation mode of REQ-ELICIT. Objects of classes Requirement and Document cannot be registered in REQ-SPEC2 because these classes are declared in REQ-SPEC2 for object use only. However, objects of these classes that are registered in REQ-ELICIT may be used.

```
tmod REQ-SPEC2 is
  use REQ-ELICIT .
  tclasses Spec Specify .
endt
```

Observe that the classes Extract and Derive are declared as private in REQ-ELICIT. Therefore, in both modules, REQ-SPEC1 and REQ-SPEC2, objects of these classes can be neither registered nor used. □

Example 44 The modules REQ-SPEC1 and REQ-SPEC2 of the previous example are changed to explicitly declare the class Requirement. Therefore, objects of this class can be registered in both modules.

```
tmod REQ-SPEC1 is
  import REQ-ELICIT .
  tclasses Requirement Spec Specify .
endt

tmod REQ-SPEC2 is
  use REQ-ELICIT .
  tclasses Requirement Spec Specify .
endt
```

In the case of REQ-SPEC1 the explicit declaration of Requirement makes no difference because the importation of REQ-ELICIT already allows the registration of requirements objects. However, in module REQ-SPEC2 the importation of REQ-ELICIT only allows the use of already created requirements. The explicit declaration of Requirement supersedes this. □

3.2.3 Transitive Module Importation

Module importation is transitive. However, the nature of a transitively imported class — if declared for object creation or for object use only — depends on the sequence of the importing declarations. Importing a module by means of an **import** declaration causes the classes declared for object creation in the imported module to be also declared for object creation in the importing module. It does not change the status of the classes declared for object use in the imported module: they remain declared for object use in the importing module. On the other hand, importing a module by means of a **use** declaration causes all classes declared in the imported module to be declared for object use only in the importing module.

Example 45 The module REQ-ELICIT is declared as in Example 43. The module REQ-SPEC1 imports REQ-ELICIT and therefore objects of classes Requirement and Document may be registered in it. Also, objects of these classes registered in REQ-ELICIT may be used in REQ-SPEC1.

```
tmod REQ-SPEC1 is
  import REQ-ELICIT .
endt
```

The module REQ-SPEC1A below imports REQ-SPEC1 by means of a use declaration, and therefore objects of classes Requirement and Document cannot be registered in it. However,

objects of these classes registered in REQ-ELICIT as well as those registered in REQ-SPEC1 may be used in REQ-SPEC1A.

```
tmod REQ-SPEC1A is
  use REQ-ELICIT .
endt
```

The sequence of the importing declarations is reversed (first use, second import) in the following modules:

```
tmod REQ-SPEC2 is
  use REQ-ELICIT .
endt

tmod REQ-SPEC2A is
  import REQ-SPEC2 .
endt
```

Although module REQ-SPEC2A imports REQ-SPEC2 preserving the classes declared for object creation, no objects of classes Requirement and Document can be created in REQ-SPEC2A. This is because these classes are declared in REQ-SPEC2 for object use only. □

3.2.4 Selective Module Importation

A module may also selectively import classes from another module. This is done by means of **from** declarations exemplified below, where the module DESIGN imports the class FuncReq from module REQUIREMENT and the classes Manual and Interview from module DOCUMENT.

```
tmod DESIGN is
  from REQUIREMENT import FuncReq .
  from DOCUMENT use Manual Interview .
endt
```

The keyword **import** after the name of the imported module indicates that the classes following it are imported for object creation. In a similar way the keyword **use** indicates that the declared classes are imported for object use.

The same considerations about classes declared for object creation and for object use are valid in this mode of module importation. In particular, it is also possible to transitively import classes in this selective way. The only difference from importing all classes of a module is that the classes of the imported module not listed in the selective import declaration are left outside the signature of the importing module. A class cannot be imported in two different ways into a single module.

Example 46 The module REQ-SPEC1 below has two selective importing declarations. The first one preserves the class declarations of the imported module. This means that objects of class Requirement may be registered in REQ-SPEC1 because the class Requirement is declared for object creation in REQ-ELICIT. The second import declaration restricts the class declarations of the imported module. This means that, although Document is declared for object creation in REQ-ELICIT, no object of this class may be registered in REQ-SPEC1.

```
tmod REQ-SPEC1 is
  from REQ-ELICIT import Requirement .
  from REQ-ELICIT use    Document .
  tclasses Spec Specify .
endt
```

The module REQ-SPEC2 selectively imports the class Requirement from REQ-ELICIT. The class Document is not imported and therefore it is not part of the signature of REQ-SPEC2: objects of class Document can neither be registered nor used.

```
tmod REQ-SPEC2 is
  from REQ-ELICIT import Requirement .
  tclasses Spec Specify .
endt
```

□

3.3 Registering Objects in TOOR Modules

All objects in TOOR are created in a unified database and they may be accessible to every TOOR module depending on the module signature. Although the actual creation of objects is centralized they may only be created in TOOR if *registered* in the scope of some module. Every object is considered to be attached to some TOOR module — the module in which it is registered. An object may only be registered in one module but it may be used in any other module provided that the module in which the object is to be used properly imports the module to which the object is attached. Of course, an object may only be registered in a module if its class is declared for object creation in that module. Also, an object may only be used in other modules if its class is not declared as private in the module in which it is registered. The registration of objects in a module is automatic and a direct result of object creation. Every object created in TOOR is registered in the current module. At any time TOOR has a module considered to be the current one. The user may change it by selecting an option in the *Module Menu* in the main window of TOOR.

3.3.1 Declaring Registered Objects

When an object is created in TOOR it is automatically registered, by means of a **retain** declaration, in the current module. This declaration consists of the class of the object and its identification, as exemplified in the module below which declares the objects `req1` and `req2` of class `Requirement` and the relation instance `der12` of class `Derive`.

```
tmod REQ-ELICIT is
  tclasses Requirement Derive .
  retain Requirement : req1 req2 .
  retain Derive : der12 .
endt
```

Relation instances are also objects and are declared accordingly. However, there is an additional constraint for a relation instance to be created in TOOR: not only should the relation class be declared for object creation but also the related objects should be visible in the current module, i.e., their classes should be part of the module's signature and it should be possible to use both objects. Thus, to relate objects registered in different modules one module should import the other in such a way that the classes of both objects may be used in the importing module.

TOOR automatically creates identifiers for relation objects. This facilitates the creation of relation instances because the user does not need to think of an identifier every time two objects are related. Relation identifiers are usually hidden from the user. Although relation identifiers are used to declare relation instances in TOOR modules, this is not a problem because, as we said before, the textual form of TOOR modules is just an internal representation. The user can always refer to a relation instance as a pair of related objects of a given class. To facilitate the understanding of our examples we, from now on, write relation instances as pairs of related objects. This is just a notational convenience.

Example 47 The relation instance `der12` registered in the module `REQ-ELICIT` above would be declared as

```
tmod REQ-ELICIT is
  tclasses Requirement Derive .
  retain Requirement : req1 req2 .
  retain Derive : (req1,req2) .
endt
```

The pair `(req1,req2)` uniquely identifies the relation instance `der12` because no relation class can have two different instances relating the same objects. □

Example 48 The following declarations are used to register in module `RSD-SPEC` the creation of requirements `req1`, `req2`, `req3`; documents `docA`, `docB`; and a relation `Extract` between `docA` and `req1`.

```
tmod RSD-SPEC is
  tclasses Requirement Document .
  tclasses Derive Extract .
  retain Requirement : req1 req2 req3 .
  retain Document : docA docB .
  retain Extract : (docA,req1) .
endt
```

□

The declaration of objects in TOOR modules just indicates the attachment of the objects to the modules, i.e., they document that objects are registered in specific modules. In particular, they do not specify objects or any of their attributes. Objects are created through templates in a unified database and are checked with respect to the contents of their attributes against the specification for their classes as defined in the project specification.

3.4 Extended Example

Before proceeding with the description of TOOR module features we present an extended example showing how a project may be structured and how objects are created and related in the context of its modules. This example also illustrates most of the remaining features of TOOR modules. It shows the specification of a bank system from the analysis of its initial requirements to the development of its FOOPS code. It is based on similar examples presented in [61, 123, 129]. In Particular, the FOOPS code specifying a solution is adapted from [123, 129]. We modify these previous versions of the example by providing an initial set of requirements and a context that explains where the requirements come from and the process by which they result in the given solution. The example serves to illustrate the use of TOOR modules and the automatic generation of relations based on the FOOPS information.

3.4.1 The Development Environment

The environment for this example consists of four main modules: `DOCUMENT` for document objects, `DD` for data dictionary (concept) objects, `REQUIREMENT` for requirement objects, and `SPEC` for specification objects. The modules are created in advance so that the analysts have only to register the desired objects in the appropriate ones. The modular structure of the project is given by

```
tmod DOCUMENT is
  tclass Document .
endt
tmod DD is
  use DOCUMENT .
  tclasses Concept BasedOn .
endt
tmod REQUIREMENT is
  use  DOCUMENT .
  from DD use Concept .
  tclass   Requirement .
  tclasses Extract  Derive .
endt
tmod SPEC is
  from REQUIREMENT use Requirement .
  from DD use Concept .
  tclasses FOOPSmod UseDomain ProtectMod ExtendMod Specify .
endt
```

Objects of class `Document` may be registered only in `DOCUMENT`. Because the module `DOCUMENT` is imported into `DD` and `REQUIREMENT`, document objects are available for use in these modules as soon as they are created. The relation class `BasedOn`, declared in `DD`, is intended to relate concept objects to the documents on which their descriptions are based. The module `REQUIREMENT` imports `DD` but only to use the class `Concept`, therefore only concept objects registered in `DD` may be used in `REQUIREMENT` and not, for example, relation instances of class `BasedOn`. In the same way the module `SPEC` may use objects of class `Requirement` from `REQUIREMENT` and objects of class `Concept` from `DD`. The other classes declared in `SPEC` are related to the use of FOOPS to specify the system. They are explained during the development of the example. The picture could be more complex showing people, design diagrams, and other types of objects but this level of detail is sufficient for the purpose of this example.

3.4.2 Eliciting Requirements

We begin with two documents. The *account description* discusses the need for new kinds of accounts and describes their basic characteristics. The *account operations* provide details of account related operations. Extracts of these documents are shown in Figure 3.3. They are registered in TOOR as objects of class `Document` identified by `acc-desc` and `acc-oper`, respectively.

From an initial analysis of the documents the following terms are selected as possibly reflecting valuable concepts for the application domain:

Account Description

Basic accounts offer only the standard facilities of deposit and withdrawal of money. They are intended for customers not legally or otherwise entitled to check books. Check accounts are basic accounts with check book facilities. Overdraft accounts allow the account holder to withdraw more than the account balance, up to a fixed limit established by the finance department. The limits for overdraft accounts are established on a monthly basis. Interest accounts pay interest rates on the account balance. The rates are also established monthly by the finance department. These facilities are cumulative so that an overdraft account holder is also entitled to check books and an interest account holder is entitled to both check books and overdraft facilities.

Account Operations

The balance of an account is the result of subtracting the debits from the account balance and adding its credits. The balance should be updated on a transaction basis each transaction consisting of the account name, the sum, and the nature of the transaction – if debit or credit. Credit transactions are restricted to deposits and automatic payments of interests. Debit transactions are restricted to withdrawals and checks. All transactions should be registered for a minimum period of one year for each account. The transaction control should show the date of the transaction, its nature, and its value. Checks should be subject to a more detailed control which includes the check number.

Figure 3.3: Extracts of Documents `acc-desc` and `acc-oper`

basic account	check register	overdraft limit	credit
deposit	overdraft account	interest account	transaction
withdraw	account holder	interest rate	transaction register
check account	account balance	debit	

In the process of analysis these selected terms are discussed with customers to clarify their meaning. Some of them denote similar notions, like deposit and credit, and some others are replaced by more significant ones, like account history for transaction register. The terms considered important are registered as objects of class `Concept` in the data dictionary (`DD`) module. At the same time requirements are identified, registered as objects, and related to their sources. The process continues until all parties agree to a satisfactory set of requirements. The configuration of objects and relations, and their respective modules, is shown in Figure 3.4. Only part of the `Concept` objects are shown in Figure 3.4. The requirements below are the agreed set of requirements used in this example.

`r01` All bank accounts should have a name identifying the account holder and a balance

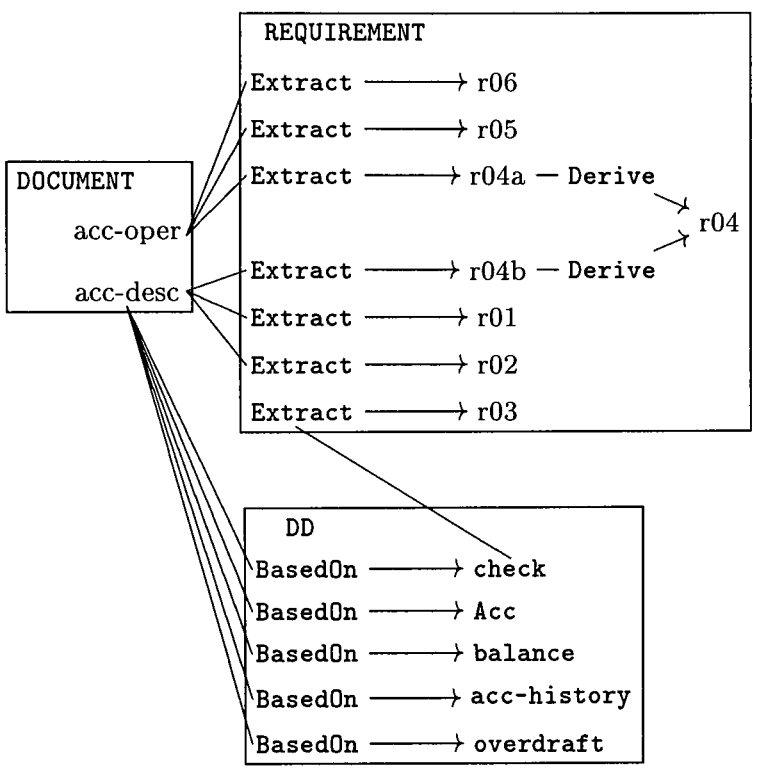


Figure 3.4: TOOR Modules and Objects at Analysis Phase

showing the amount of money currently available.

`r02` Overdraft accounts should have a limit specifying the maximum amount for overdraft.

`r03` A check should be paid only if there are sufficient funds. For overdraft accounts a check should be paid only if the sum of the balance and the limit of the account is greater than or equal to the value of the check.

`r04` Interest accounts should have an interest rate used to calculate the interest to be paid. The interest is a result of multiplying the interest rate by the account balance.

`r05` Every account should keep a history of its transactions consisting of the date of the transaction, the sum involved, and whether it is a debit or a credit.

`r06` For accounts with check books, a history of the checks should be kept containing the date, the number, and the value of the check.

Observe in Figure 3.4 that requirement `r04` is registered as a derivation of two previously registered requirements and that requirement `r03` is related to (the description of) the concept object `check`.

3.4.3 Specifying the System

The specification content of TOOR modules is indirectly determined by the specification contents of their objects. One particular class of objects having a definite specification content is that of FOOPS related objects. TOOR is specially designed to use FOOPS as the specification language for software development, and its close connection with FOOPS provides a smooth integration between the software development structure (given by the FOOPS modules) and the project development structure (given by the TOOR modules).

3.4.3.1 Creating FOOPS Module Objects

A FOOPS module is just an object in TOOR. It has a pre-defined class called `FOOPSmod` that does not need to be defined in the project specification. Its template is shown in Figure 3.5. The field after the keyword *omod* contains the name of the module. The next two fields are for the module interface and renaming declarations, respectively. The field after the keyword *is* contains the specification of the body of the module. If a FOOPS module refers to built-in functions they may be written in a separate window that pops up when one of the buttons *Built-in object* or *Built-in text* is clicked on.

The template of Figure 3.5 is for a FOOPS object module, i.e., a module intended to specify object classes. It illustrates the registration of the object `ACC` which is a FOOPS module specifying a basic account in our example. Other types of FOOPS modules like functional modules and theories are also objects of the class `FOOPSmod`. They have similar templates with the keywords *fmod*/*endf* and *oth*/*endoth* substituted for *omod*/*endo*, respectively. (for details on built-in sorts and functions, and on the different types of FOOPS modules see [123, 129]).

3.4.3.2 Registering FOOPS Modules Objects

The bank account system of this example is specified using FOOPS. Each main FOOPS module is registered in a specific TOOR module created inside `SPEC`. Auxiliary FOOPS modules are grouped together in a TOOR module called `AUXSPEC`. For example, a module `MONEY` specifying a data type for currency and a module `HIST` specifying a data type for account histories are defined and registered in `AUXSPEC` as objects of class `FOOPSmod`. The first FOOPS module specifies a basic account as a class called `Acc`. It is intended to meet requirements `r01` and `r05`. These requirements state that accounts should have a name, a balance, and a history of their transactions. Therefore:

```
omod ACC is
  class Acc .
  protecting HIST .
```

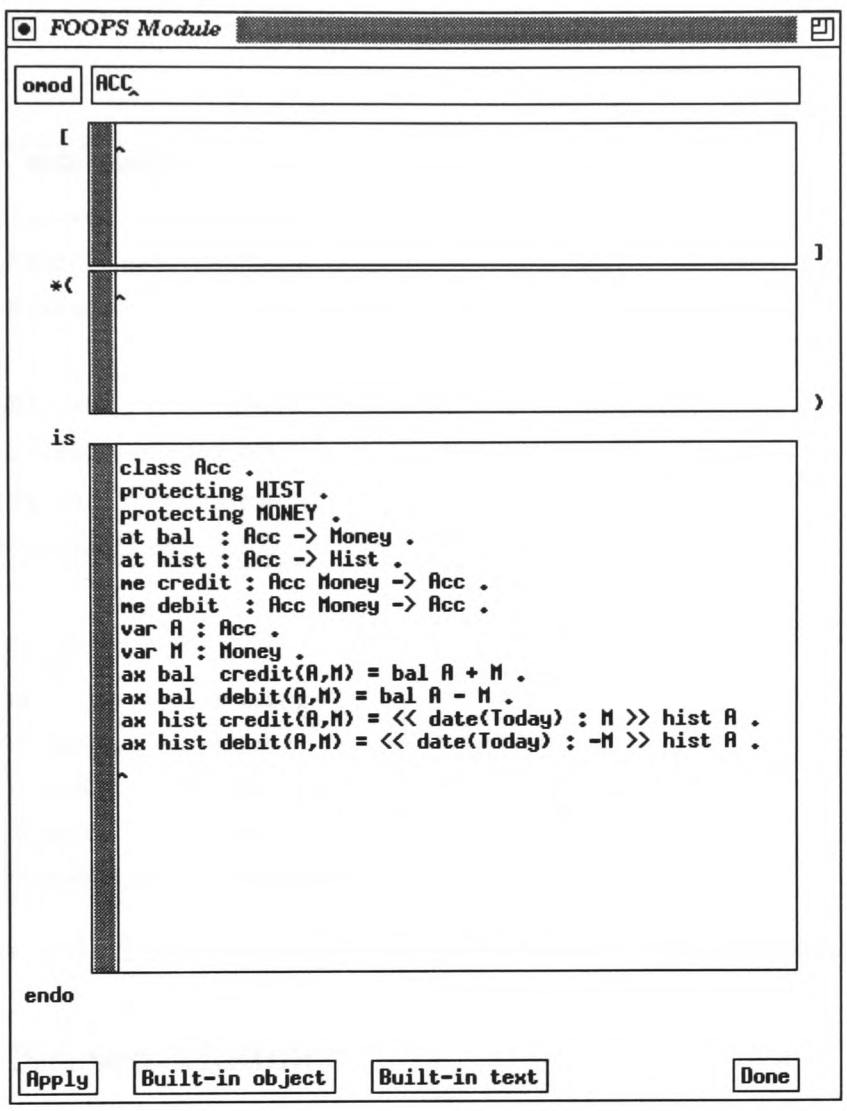


Figure 3.5: Template for FOOPS Module Objects

```
  protecting MONEY .
  at bal   : Acc -> Money .
  at hist  : Acc -> Hist  .
endo
```

The name of an account is taken to be the identification of the corresponding object of class `Acc` and does not need to be specified as an attribute. The sorts `Money` and `Hist` are specified in the modules `MONEY` and `HIST`, respectively. These modules are imported into `ACC` (by `protecting`) so the data types in there may be used. The FOOPS module `ACC` itself is registered as a `FOOPSmod` object in the TOOR module `ACCOUNT`, and is related to requirements `r01` and `r05` through instances of the `Specify` relation. The module `ACCOUNT`, after the relations to requirements are made, is given by

```
tmod ACCOUNT is
  retain FOOPSmod : ACC .
  retain Specify  : (r01,ACC) (r05,ACC) .
endt
```

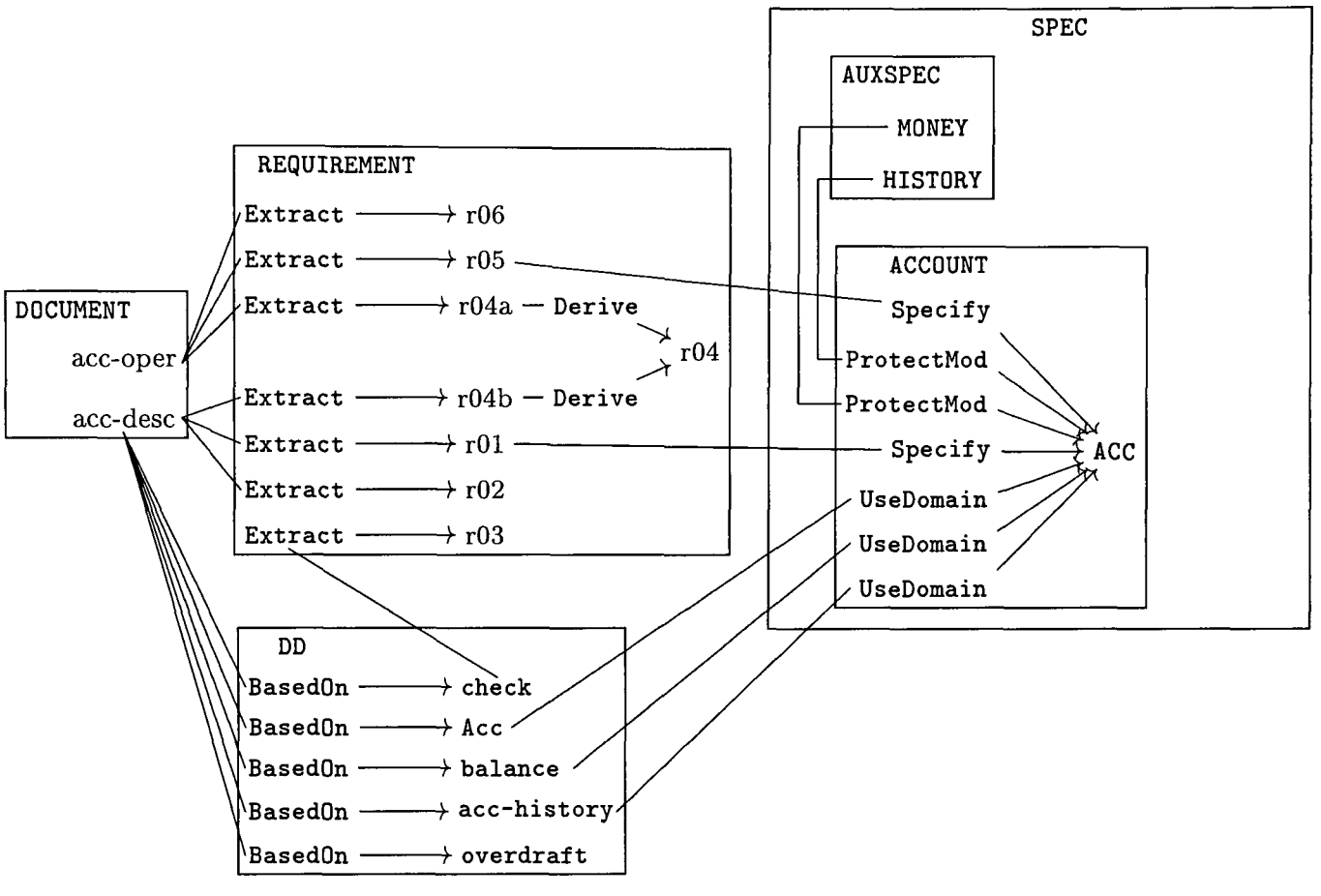


Figure 3.6: TOOR Modules and Objects at Specification Phase

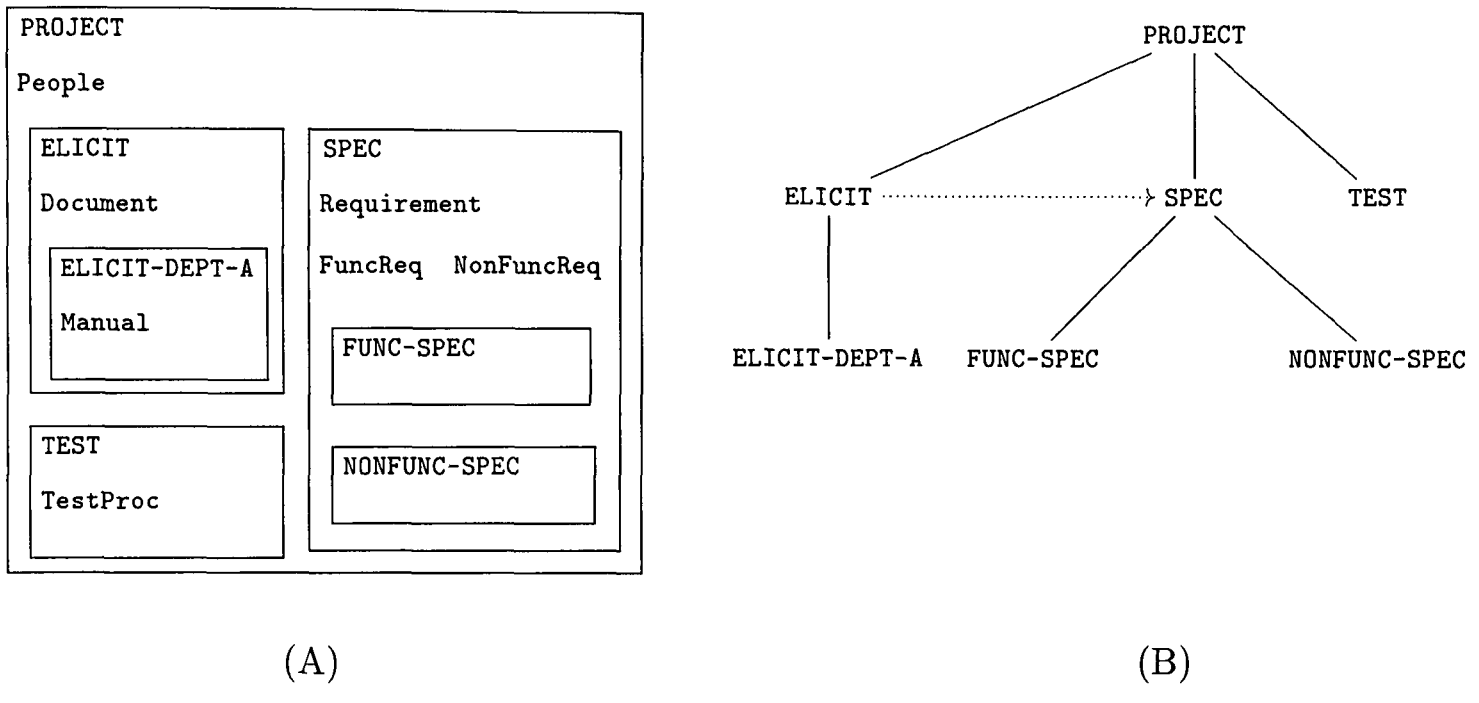
3.4.4 Creating Nested Modules

The TOOR modules `AUXSPEC`, and `ACCOUNT` are created inside the TOOR module `SPEC` as *nested modules*. The textual form of `SPEC`, after creation of these nested modules, is given by

```
tmod SPEC is
  from REQUIREMENT use Requirement .
  from DD use Concept .
  tclasses FOOPSmod UseDomain ProtectMod ExtendMod Specify .
  tmod AUXSPEC is
    retain FOOPSmod : MONEY HIST .
  endt
  tmod ACCOUNT is
    retain FOOPSmod : ACC .
    retain Specify  : (r01,ACC) (r05,ACC) .
  endt
endt
```

and the module structure of the project, up to this point, is shown in Figure 3.6.

Creating a module inside another one makes the objects of the inner module invisible to the outer module while maintaining all objects of the outer module visible to the inner module. The classes declared in the outer module are considered to be declared in all inner modules. This is why the modules `AUXSPEC` and `ACCOUNT` do not contain class declarations. The declarations in `SPEC` are valid for them. If an inner module contains no such declarations



(A) (B)

Figure 3.7: Module Importation in TOOR

its signature is taken to be the same as the signature of its immediate outer module. In any case the signature of an outer module is always a subsignature of its inner modules. This is enforced by TOOR: every class in the signature of an outer module, explicitly declared or imported from other modules, is also considered to be part of the signature of its inner modules.

3.4.4.1 Importing Nested Modules

We now make a small deviation from our example to explain the effect of module importation on nested modules. A module with nested modules exports only the classes declared in its own signature and none of the classes declared in the signatures of its inner modules. For instance, in Figure 3.7(A) modules are shown as boxes and nested modules as boxes inside boxes. Each module shows the module name, in capitals, and the name of its declared classes. Then, any module importing `PROJECT` incorporates in its signature only the class `People`. The converse — importing a module into another module containing nested modules — makes the imported classes part of the signature of the importing outer module and also part of the signature of all its inner modules. For instance, if in Figure 3.7(A) the module `SPEC` imports the class `Document` from module `ELICIT`, this class is incorporated into the signature of `SPEC` as well as into the signatures of `FUNC-SPEC` and `NONFUNC-SPEC`.

A module can neither import (classes from) its own nested modules nor import (classes from) modules that are inner modules to its nested modules. An inner module cannot import its own outer module. More generally, a nested module can only import another module if both are inner modules of the same outer module.

Example 49 In Figure 3.7(A) we have that (*i*) the module PROJECT cannot import any module because all other modules are its inner modules, (*ii*) the modules ELICIT, TEST, and SPEC can import each other because they all have the same immediate outer module, namely PROJECT, (*iii*) the module ELICIT-DEPT-A cannot import any module because no module is also an inner module of ELICIT. □

The constraints on importation of nested modules preserves the original effect of creating objects inside them: if a module B is created inside a module A, the objects in B are invisible to A and, therefore, they should not be visible to any module importing A. Nevertheless, this restriction may be overcome by explicitly importing the desired module through a sequence of importing declarations. If a module C imports a module A which has a nested module B, then, after importation, the module B is also regarded as an inner module of C. This *transference* is unidirectional. It works as if the inner modules of the imported module were also declared as inner modules of the importing module.

Example 50 Figure 3.7(B) presents the modules of Figure 3.7(A) in a tree-like form showing the hierarchy of the nested modules. If SPEC imports ELICIT (indicated by a dotted arrow), then both FUNC-SPEC and NONFUNC-SPEC may import ELICIT-DEPT-A because, after importation of ELICIT they are seen as having the same outer module. Note that in this situation module ELICIT-DEPT-A can import neither FUNC-SPEC or NONFUNC-SPEC. □

3.5 Automatic Generation of Traces

TOOR has special functions to deal with FOOPS modules, FOOPS built-ins, FOOPS hierarchies, and other FOOPS features. By making use of the precise form of FOOPS texts, TOOR is able to automatically produce trace information related to these texts.

3.5.1 Automatic Traces of Structural Information

TOOR automates the module structure of FOOPS by providing relations to specify the dependencies among FOOPS modules. If one FOOPS module imports another, TOOR automatically relates the corresponding FOOPS module objects by means of relations that characterize the importing modes of FOOPS: **extending**, **protecting**, and **using**. The respective TOOR relation classes for these importation modes are **ExtendMod**, **ProtectMod**, and **UseMod**. We proceed with the extended example of Section 3.4 to illustrate this automatic generation of traces.

Requirement **r06** says that every check account should keep a history of its check transactions. Therefore, the class **ChAcc** for check accounts is specified with an specific attribute

chist. The specification of check accounts is given by the following FOOPS module which also defines a method (debtcheck) and an axiom showing how checks are debited and included in the check history. The particular form of this axiom specifies a method composition (indicated by the semicolon) but it is not of interest here.

```
omod CHACC is
  class ChAcc .
  extending ACC .
  subclass ChAcc < Acc .
  protecting CHIST .
  at chist : ChAcc -> Chist .
  me debtcheck : ChAcc Check -> ChAcc .
  var A : Acc . var C : Check .
  ax debtcheck(A,C) = debit(A,value(C)) ; C chist A .
endo
```

This module imports the module ACC to declare ChAcc as a subclass of Acc. This is in accordance to requirements r01 and r05: the features already specified for Acc should be features of every type of account. The FOOPS module CHACC is registered as a FOOPSmod object in TOOR module CHECK-ACC:

```
tmod CHECK-ACC is
  import ACCOUNT .
  retain FOOPSmod : CHACC .
endt
```

At any time the user can select in the *trace menu* an option to automatically generate trace information based on the content of FOOPS objects. This option, given a FOOPS module, check each FOOPSmod object in it: for each FOOPSmod object A containing a declaration importing B, a relation instance is created to relate A and B. The relation class depends on how B is imported into A. Of course, B should exist as a FOOPSmod object and should be visible to the TOOR module in which A is registered.

In our example, the user selecting this option in the context of module CHECK-ACC causes the creation of a relation instance of class ExtendMod relating the object ACC to the object CHACC . The TOOR module CHECK-ACC becomes

```
tmod CHECK-ACC is
  import ACCOUNT .
  retain FOOPSmod : CHACC .
  retain ExtendMod  : (ACC,CHACC) .
endt
```

The automatic generation of relations between FOOPS module objects takes into account the module composition features of FOOPS. For example, if a FOOPS module `mod-A` contains the declaration `protect mod-B + mod-C`, specifying that it imports the composition of `mod-B` and `mod-C`, then TOOR automatically relates `mod-A` to the FOOPS module objects corresponding to `mod-B` and `mod-C`. TOOR also automatically creates relations to reflect the use of FOOPS modules as parameters of generic FOOPS modules. The pre-defined relation class provided for this case is called `ViewMod`.

In FOOPS an instantiation of a generic module is usually declared inside other FOOPS modules as part of a module importation command. Therefore, such instantiated modules are not registered as objects in TOOR. They are rather part of other FOOPS module objects.

Example 51 Consider the FOOPS module below which imports an instantiation of `LIST` by `NAT`.

```
fmod LIST-NAT is
  protecting LIST[NAT] .
endf
```

Suppose that this module is registered as a `FOOPSmod` object `LIST-NAT` in a TOOR module `SYS-SPEC`. Also, the FOOPS modules objects `LIST` and `NAT` are visible to `SYS-SPEC` but there is not object identified by `LIST[NAT]`. In this situation the attempt to automatically relate by `ProtectMod` the FOOPS module objects `LIST-NAT` and `LIST[NAT]` would fail because there is no object `LIST[NAT]`. However, TOOR successfully relates `LIST` to `NAT` (by means of `ViewMod`) and uses this relation instance as the source for the `ProtectMod` relation. The textual form of the resulting TOOR module is

```
tmod SYS-SPEC is
  retain FOOPSmod   : LIST NAT LIST-NAT .
  retain ViewMod    : (LIST,NAT) .
  retain ProtectMod : ((LIST,NAT), LIST-NAT) .
endt
```

For tracing purposes this is entirely satisfactory and clearly indicates that what is being imported is not an actual FOOPS module object but rather a non-existent object that may be viewed as a relation between `LIST` and `NAT`. □

3.5.2 Automatic Traces of Domain Information

In object-oriented software development classes, attributes, and methods are usually used to represent concepts of the domain being modelled. The same may happen to data types and their operations. These concepts come from domain descriptions and may be registered in

TOOR in several ways: as part of scenario objects, interview transcript objects, statement objects, etc. They also may be registered as objects themselves. It is obvious that there is a relation between a specification text and the concepts intended to be specified by it. TOOR attempts to capture this domain information by automatically making relations between objects representing concepts and FOOPS module objects specifying them. For every sort, class, attribute, operation, and method name used in a FOOPS text, TOOR attempts to make a relation between the FOOPS object containing the FOOPS text and other objects whose identifiers are identical to the name (of the sort, class, attribute, etc.) in question. These relations are generated as instances of the pre-defined class `UseDomain`.

Below, we show again the module `ACCOUNT` of our extended example. For a notational convenience instead of just declaring the identifier of the `FOOPSmod` object `ACC` we write it in full. The elements of the FOOPS text which are declared in the module `ACC` are only the class `Acc` and the attributes `bal` and `hist`. The sorts `Money` and `Hist` are declared in the modules `MONEY` and `HIST`, respectively.

```
tmod ACCOUNT is
  retain FOOPSmod :
     omod ACC is
       class Acc .
       protecting HIST .
       protecting MONEY .
       at bal  : Acc -> Money .
       at hist : Acc -> Hist .
     endo .
  retain Specify : (r01,ACC) (r05,ACC) .
endt
```

To generate automatic traces of domain information, TOOR attempts to make relations between the FOOPS module object `ACC` and objects identified by `Acc`, `bal`, and `hist`. The user should inform the TOOR module in which the automatic relations are made; in this case the module `ACCOUNT`. TOOR searches all objects visible to `ACCOUNT` to find those identified by `Acc`, `bal`, or `hist`. We may refer to the project module structure shown in Figure 3.6 to understand the order of the search. First TOOR searches the module `ACCOUNT`, then its outer module `SPEC`, and finally the modules `REQUIREMENT` and `DD` imported by `SPEC`. Considering that the only objects registered in `DD` are those shown in Figure 3.4, the module `ACCOUNT` after the automatic generation of `UseDomain` relations, becomes

```
tmod ACCOUNT is
  retain FOOPSmod :
     omod ACC is
       class Acc .
       protecting HIST .
```

```
        protecting MONEY .
        at bal  : Acc -> Money .
        at hist : Acc -> Hist .
      endo .
  retain Specify : (r01,ACC) (r05,ACC) .
  retain UseDomain : (Acc,ACC) .
endt
```

Only the Concept object Acc, registered in DD, is found that can be automatically related to the FOOPSmod object ACC. The concepts of bal and hist are also registered in DD, but as objects balance and acc-history, respectively. This name mismatch prevents TOOR from generating automatic relations for bal and hist. Nevertheless, the user can still use TOOR to help identify this kind of missing link.

3.5.3 Tracing Missing Links

By virtue of its automatic generation of trace information, TOOR is able not only to trace objects according to the existing relations between them, but also to indicate what relations should exist but do not. This way of tracing results, for each FOOPS object, in a list containing the elements for which an automatic relation failed to be made. A check procedure indicates for each FOOPS object which of its elements should be but is not related to any object. The user initiates this check procedure by selecting an appropriate option in the TOOR's *check menu* and giving the name of the module to be checked.

At this point in the development of our extended example we can observe that there are no methods specifying how to modify the balance and history of accounts. We can trace the FOOPSmod object ACC back to its source, through its requirements, and try to find out any indication of how modifications to the values of attributes bal and hist are to be specified. However, we first check the TOOR module in which ACC is declared to see if we can get any additional hints. Tracing the missing links of ACCOUNT produces the following result:

unrelated module HIST.
unrelated module MONEY.
unrelated attribute bal.
unrelated attribute hist.

The FOOPS modules HIST and MONEY are unrelated because the TOOR module in which they are declared, AUXSPEC, is not imported into the module ACCOUNT. The class Acc is not listed because TOOR is able to make an automatic relation between the FOOPSmod object ACC and the Concept object Acc. We know that Concept objects are declared in the TOOR module DD which is imported to SPEC and therefore visible to it and to any of its nested modules. The attributes bal and hist should not have been listed as missing links. They should be

based on existing concepts and we have just seen that the module DD declaring the concepts for the system is properly imported. Therefore, we decide to inspect DD and look at its contents. The Concept objects balance and acc-history in DD are descriptions of exactly the concepts intended to be modelled by attributes bal and hist. Moreover, the description of these concepts gives the necessary information of how their values are modified by debit and credit transactions.

3.5.4 Using Automatic Tracing Facilities to Make Informed Changes

The information received from tracing the missing links of ACCOUNT and the consequent analysis are the basis for the following decisions and changes:

1. to import the module AUXSPEC into ACCOUNT,
2. to register the relation instances of class UseDomain between the FOOPS module object ACC and the concept objects balance and acc-history (these relations are made manually because we decide not to change the names of either the concepts or the attributes), and
3. to update the FOOPS module object ACC specifying the methods for debit and credit according to the information got from the description of balance and acc-history.

The module ACCOUNT, after all these actions, becomes

```
tmod ACCOUNT is
  use AUXSPEC .
  retain FOOPSmod :
     omod ACC is
       class Acc .
       protecting HIST .
       protecting MONEY .
       at bal  : Acc -> Money .
       at hist : Acc -> Hist .
       me credit : Acc Money -> Acc .
       me debit  : Acc Money -> Acc .
       var  A    : Acc .
       var  M    : Money .
       ax bal  credit(A,M) = bal A + M .
       ax bal  debit(A,M)  = bal A - M .
       ax hist credit(A,M) = << date(Today) : M >> hist A .
       ax hist debit(A,M)  = << date(Today) : - M >> hist A .
     endo .
  retain Specify : (r01,ACC) (r05,ACC) .
  retain ProtectMod : (HIST,ACC) (MONEY,ACC)
```

```
    retain UseDomain : (Acc,ACC) .
    retain UseDomain : (balance,ACC) (acc-history,ACC) .
endt
```

The fact that the concept of an account balance is registered as an object with one name and used in a FOOPS text with another one is not hard to understand. Indeed, not all concepts that are specified in FOOPS texts need to be registered as objects. They may be part of other objects and may be used as such. For example, if a requirement describing a bank account is registered it may not be necessary to register an additional object to hold the bank account concept. Also, it may be the case that some concepts are taken for granted and there would be no descriptions of them whatsoever. Even if a concept is registered as an object it may have a name different from the name used in a specification text. This may happen because different phases require different vocabulary. In a elicitation phase, where the customer is mostly involved, objects tend to have long and self-explanatory names while in specification and coding phases names tend to be more concise. For example we may have the names `bank-account-holder` and `holder` referring to the same concept. In any of these situations TOOR fails to automatically make the `UseDomain` relations. However, the user can manually register any desired relation instance using a specific template. The template for `UseDomain` has an additional field (besides the normal fields `source` and `target`) called `dom-elem`. This field is intended to contain an explanation for the connection between the related objects. For example, the `dom-elem` attribute of the manually generated instance of `UseDomain`, relating `balance` to `ACC`, may be used to identify the precise element of `ACC` on which the relation is based; it may contain, for instance, the text "balance is the concept specified by bal".

However, not all elements of a FOOPS text should exist as an object in the project environment. For example, there are some sorts that are so pervasive that it is better not to register a relation every time they are used. Others may be self-explanatory or their meaning taken for granted. Also, there may already exist relations between a FOOPS module and another object so that it is unnecessary to register another relation between them. To indicate the elements that are intentionally left outside the automatic generation of relations the user builds a list with their names. The names in this list are not considered by TOOR when tracing missing links. Natural elements of this list are the sorts `Bool` for boolean, `Nat` for natural numbers, `Id` for literal identifiers, and the class `CTriv` used to specify an arbitrary class in FOOPS.

Figure 3.8 shows the final module structure and the configuration of objects and relations of our example. The rest of the specification consists of the FOOPS modules `CHACC` for check accounts, `OVRACC` for overdraft accounts, and `INTRACC` for interest rate accounts. After a similar analysis each one is registered in a specific TOOR module and the whole system is

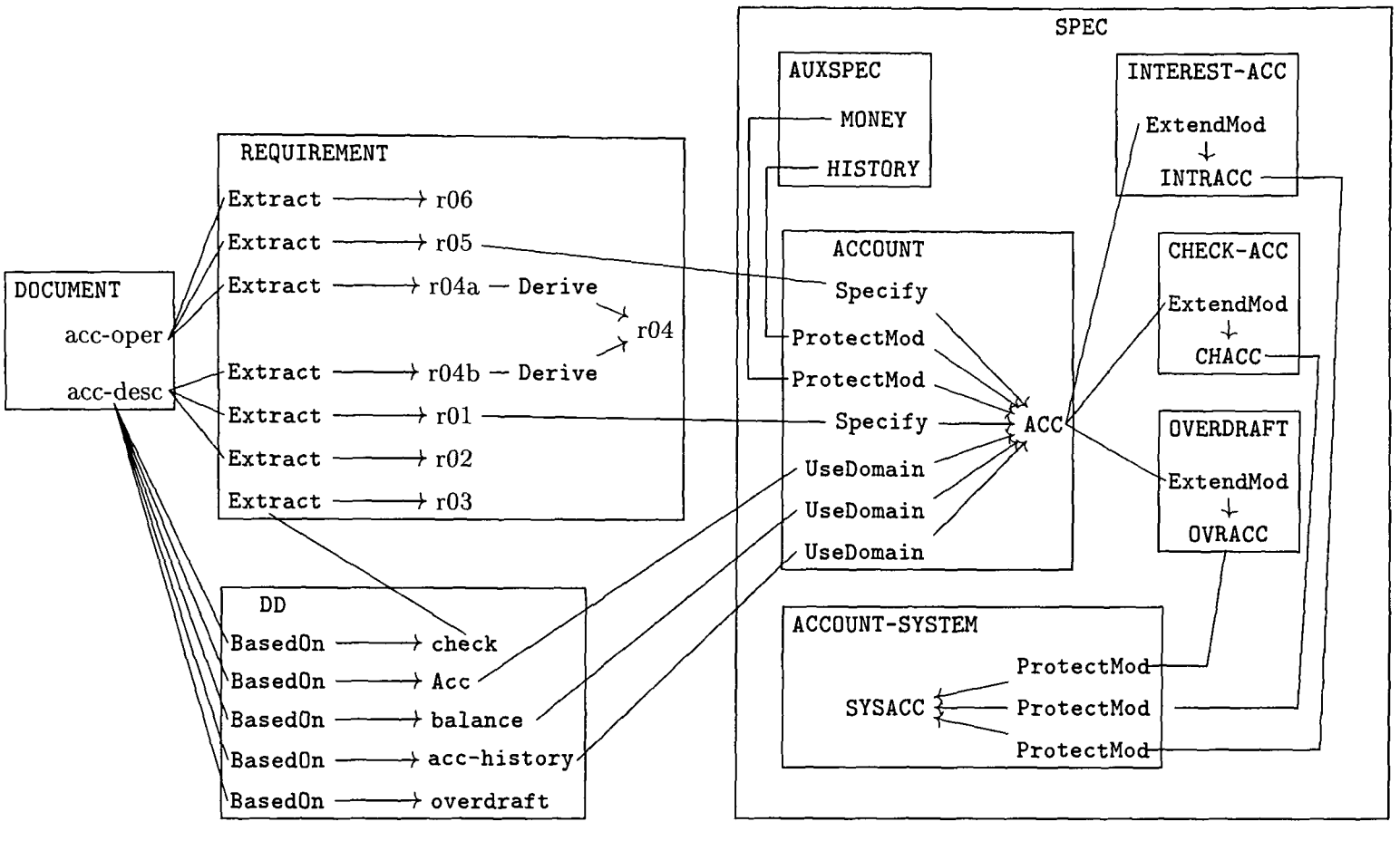


Figure 3.8: TOOR Modules and Objects at Completion of Specification

specified by a composition of these FOOPS modules:

```
omod SYSACC is
  protecting ACC + CHACC + OVRACC + INTRACC .
endo
```

which is also registered as an object.

3.6 Other Aspects of Using FOOPS in TOOR

The `FOOPSmod` objects of our extended example were directly registered in TOOR using appropriate templates. However, FOOPS is an independent system and it may be the case that a specification is developed entirely within the FOOPS system. Even in this case it is still possible to use TOOR to trace FOOPS information. TOOR can read FOOPS files and either register an entire file as a special kind of object or register each FOOPS module in the file as a `FOOPSmod` object. In the reverse way, TOOR can generate FOOPS files from the information contained in its registered `FOOPSmod` objects.

3.6.1 Registering General FOOPS Texts

FOOPS objects can also be registered in TOOR as instances of the class `FOOPStext`. This class is intended to hold general FOOPS texts and its objects are texts containing any FOOPS

commands. It is useful when some FOOPS specification containing several FOOPS modules and other FOOPS declarations may be treated as a unit. In such cases it is more efficient to register the entire FOOPS text as one object, instead of registering each module and built-in function in it as a separate object. It also may be used to register test procedures in the form of FOOPS commands to evaluate previously defined FOOPS objects. The template for `FOOPStext` contains just two fields: the first is for the identification of the object and the second is for the FOOPS text.

3.6.2 Using Previously Written FOOPS Files

FOOPS specifications are usually written in a file containing several FOOPS modules. Although TOOR has the special class `FOOPStext` to hold general FOOPS texts, it is sometimes convenient not to register the entire FOOPS text as a single object. The user can use an external file containing a FOOPS specification to directly create `FOOPSmod` objects in TOOR. In this case every FOOPS module in the file becomes a `FOOPSmod` object and TOOR automatically makes relations between them.

3.6.3 Generating FOOPS Specifications

A FOOPS specification may be extracted from any TOOR module. The extraction procedure generates a file containing all FOOPS texts registered as objects in the module. This file can then be submitted to execution in a FOOPS session. This provides a useful way to use the executable features of FOOPS to test the system being specified in TOOR. For example, extracting the FOOPS specification of the TOOR module `ACCOUNT-SYSTEM` of our extended example produces a FOOPS specification text containing the module `SYSACC` together with modules `OVRACC`, `CHACC`, `INTRACC`, `ACC`, `MONEY`, and `HIST`. This specification can then be tested in a FOOPS session using the executable features of the FOOPS system. Such a test might show that the specification of our example does not adequately meets its requirements. For example, the module `CHACC` specifies the method `debtcheck` as a debit of the check value to the account balance, followed by an inclusion of the check in the check history of the account:

```
me debtcheck : ChAcc Check -> ChAcc .
var C : Check .
var A : ChAcc .
ax debtcheck(A,C) = debit(A,value(C)) ; C chist A .
```

However, it does not consider the sufficient amount of funds mentioned in requirement `r03`. Also, the requirements themselves do not correctly reflect what is expected from the system. For example, document `acc-desc` says that the account facilities should be cumulative and

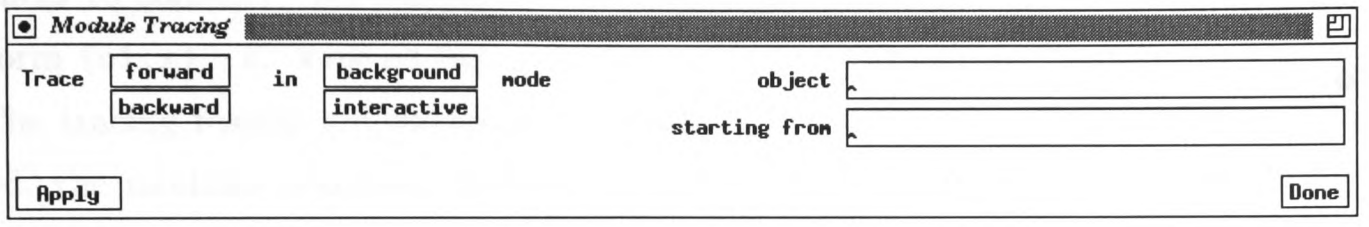


Figure 3.9: Module Tracing Template

there is no requirement expressing it. Of course, requirements and specification can be modified. In the process of modifying them new needs for tracing objects are certain to arise and the modular structure of the project may be used to guide or restrain the tracing.

3.7 Tracing with Modules

TOOR modules, besides the facilities to automatically generate trace information, are used for tracing purposes in two ways. First, they provide a context restricting the tracing space. In this way a tracing procedure may consider only objects that are in a given module, or set of modules. Second, the module structure itself can be used to trace an object forward or backward. This is done by selecting the object to be traced, the module in which to start the tracing, and the direction of tracing. The result containing all objects and relations is stored in a file that may be browsed. Alternatively, the tracing may be done interactively. In this case, the search tree is shown interactively for each module and the user commands the next step in the search. The user can alter the order of the search, interrupt the search to inspect some object, and backtrack to previously visited objects.

The *Module Tracing* template of Figure 3.9 shows the options available to use the module structure to guide the trace. The basic way is to give an object identifier, the direction of tracing, and how the trace should be performed, either interactively or in background mode sending the results to a file. The user may also choose a specific module to start the trace; if no module is given, the trace starts from the module where the object is registered.

3.7.1 Interactive Tracing

For an object `o1` registered in module `Mod-A` TOOR searches the relation instances declared in `mod-A` which involves `o1`. The search traverses the module structure to consider relation instances registered in modules created inside `mod-A`, and in modules importing `mod-A`. For each relation instance `(x,o1)` TOOR keeps note of the related object `x` and when the search is finished for `o1` TOOR repeats the process, now searching for relations involving each of the objects `x` obtained in the previous step. The direction of tracing indicates which relation

instances to consider. For example, a forward tracing of `o1` considers relation instances of the form `(o1,x)`, i.e., with `o1` being the first object of the pair.

The tracing results are shown step by step in a window that shows the objects already traced, the modules searched, the objects to be traced next, the modules to be searched next, and the result so far. At each step the user may choose to skip the step, to proceed either in the normal tracing order or selecting another object or module from the list of the next ones, or to backtrack to a previous step. The user may also inspect the contents of an object by clicking on its entry in the list of traced objects or in the list of the next objects to be traced. This may give additional information to choose the next module to be searched, to proceed with the tracing in a different order, or to stop the tracing and start a new search. Figure 3.10 shows the partial result of tracing forward the document `acc-desc` of our example. We may refer to Figure 3.8 to understand the steps taken so far. First, the module `DOCUMENT` is searched and no relations involving `acc-desc` are found. Then, the modules `REQUIREMENT` and `DD` are placed in the list of modules to be searched next. In searching the module `REQUIREMENT`, the objects `r04b`, `r01`, and `r02` are found to be related to `acc-desc`. The relation instances are placed in the *Result* window and the related objects in the window showing the next objects to be traced. Also the module `SPEC` is added to the list of the modules to be searched and the module `REQUIREMENT` is removed from this list and put in the list of already searched modules.

3.7.2 Background Tracing

Modular tracing in background mode causes the result of tracing to be sent to a file. The direction of tracing cannot be changed. The entries in the resulting file show the relation instances traced, the related objects, and the modules where the relation instances are registered. As an example, tracing forward, in background mode, the document `acc-oper` of our extended example results in a file containing the following entries:

```
(REQUIREMENT, (acc-oper,Extract,r06)), (REQUIREMENT, (acc-oper,Extract,r05)),
(REQUIREMENT, (acc-oper,Extract,r04a)), (REQUIREMENT, (r04a,Derive,r04)),
(ACCOUNT, (r05,Specify,ACC)), (INTEREST-ACC, (ACC,ExtendMod,INTRACC)),
(CHECK-ACC, (ACC,ExtendMod,CHACC)), (OVERDRAFT, (ACC,ExtendMod,OVRACC)),
(ACCOUNT-SYSTEM, (INTRACC,ProtectMod,SYSACC)),
(ACCOUNT-SYSTEM, (CHACC,ProtectMod,SYSACC)),
(ACCOUNT-SYSTEM, (OVRACC,ProtectMod,SYSACC)).
```

This can be printed as

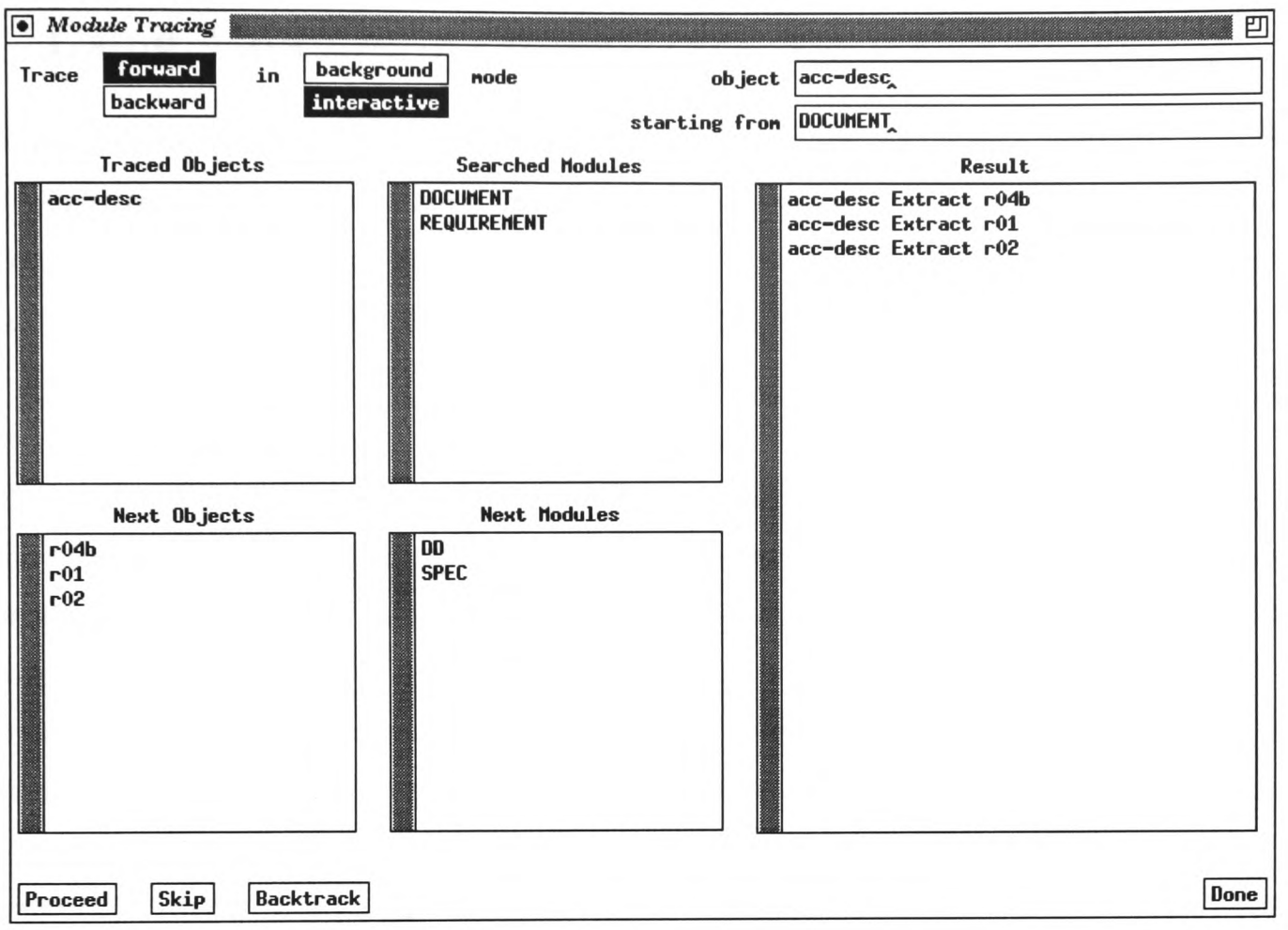


Figure 3.10: Interactive Tracing Window

```
in REQUIREMENT   object acc-oper relates   to r06    by Extract
                                           to r05    by Extract
                                           to r04a   by Extract
                 object r04a relates       to r04    by Derive
in ACCOUNT       object r05 relates        to ACC    by Specify
...
```

The rest of this printed version has been omitted.

3.7.3 Specifying a Starting Module

Giving a module to start the tracing is useful when one wants to leave some modules out of the tracing procedure. For example, if the module structure for the project is linear, resembling the waterfall life cycle model, then one may trace a requirement forward starting from the module associated to the design phase. This leaves out of the trace all information contained in the previous module associated to the requirements phase. The result of tracing an object from a given module is always a subset of tracing the same object from the module where it is registered.

3.8 Using Modules in Other Trace Modes

TOOR modules are used to structure the development by providing contexts for objects. Whatever structure is chosen for a project the same structuring mechanisms may be used to provide a context for tracing. For example, if a question about the validity of some requirement arises and the user wants to trace the requirement following its relations to other requirements and to documents, then the tracing may be restricted to the module where requirements and documents are registered. This is particularly useful if the project is well advanced in its development, e.g., if the system is being coded. In this case, there may be several other objects directly or indirectly related to the requirement in question such as specification texts, test cases, source code, etc., that may not be of interest in the first instance.

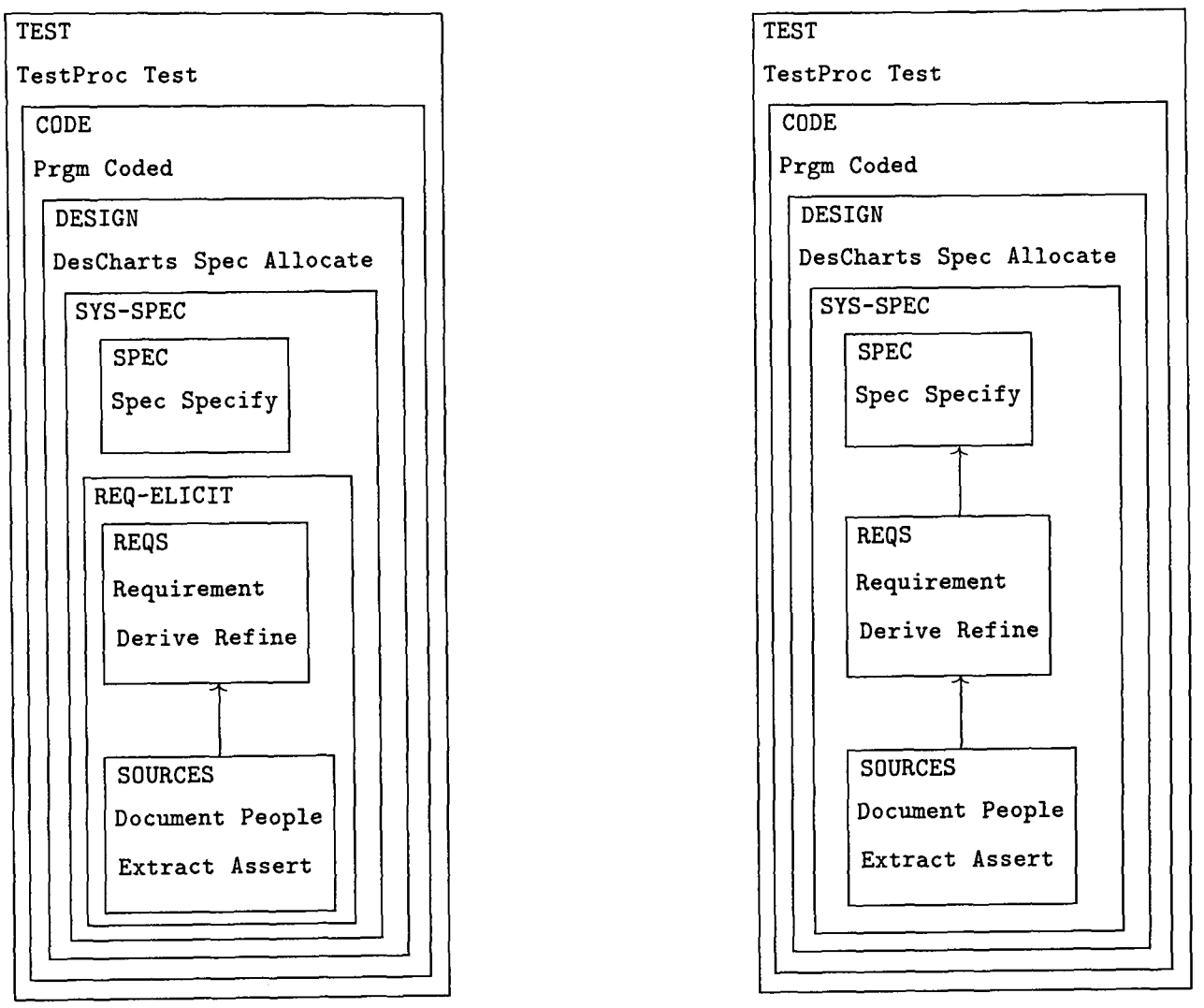
TOOR has different modes of tracing. In each mode it is possible to restrict the tracing procedure to a certain TOOR module. One way of tracing objects in TOOR is to browse them using graphical interfaces to show the objects and their relationships. The user may choose to show only objects of a given module or group of modules. This is explained in Chapter 5. Another way of tracing objects in TOOR is through the use of regular expressions. The user may restrict the pattern matching procedure to objects created in certain module or group of modules. This is explained in Chapter 4.

3.9 A Note on Method

TOOR does not enforce any policy on the use of modules. Although one should use the ability to import modules and to create nested modules according to some criteria, the enforcement of a module policy is a methodological point outside the scope of this thesis. Nevertheless, some comments are in order.

The first point is that it is not necessary to structure a project at all. Every project in TOOR has an automatically defined module, called `PROJECT`, whose signature consists of all classes declared in the FOOPS module `TOORFRONTEND` of the project specification. Therefore, objects of all classes declared in `TOORFRONTEND` may be created and related in `PROJECT`. The module `PROJECT` is used as a default for the current module. In its most simple form a project may be developed entirely within the module `PROJECT`.

TOOR modules may be used to structure the project as a whole or to organize parts of it, e.g., grouping related objects and defining a policy for object creation and use. Nested modules may be used to circumscribe objects created as a result of tentative trials. The objects created in a nested module are not visible from outer modules and therefore do not



(A) (B)

Figure 3.11: Structuring a Project According to the Waterfall Life Cycle Model

interfere with the normal development path. For example, one may create inside a REQS module, serving as a scope for requirements objects, a module TENTATIVE-REQ to be used to experiment with requirements derivation and refinement. When the requirements and relations of TENTATIVE-REQ are considered satisfactory the nested module TENTATIVE-REQ may be removed and its objects incorporated into REQS. The danger in using deep nested structure is to make the project as a whole highly hierarchical, making it difficult to have a broad idea of what is going on. On the other hand, to make no use of modules, i.e., to create all objects in the scope of PROJECT can result in a highly fragmented view of the project, consisting entirely of its objects and relations.

A project can be nicely structured by making modules import each other in a way that results in an effective separation of concerns with respect to its various phases or stages. Figure 3.11(A) shows part of a possible module structure for a project development according to the waterfall life cycle model. Using this structure documents, people, and requirements objects together with relations among them are created in module REQ-ELICIT intended to be the scope of the requirements elicitation phase. At the completion of this phase the module

REQ-ELICIT is removed resulting in the situation shown in Figure 3.11(B). The module SPEC imports REQS to relate requirements objects to specification objects. The passage to the next phase is similar. Other module structures may be created to resemble different development processes, e.g., prototyping and evolutionary models.

It should be noted that a module structure may be created in advance by a manager or a project leader. Other TOOR users like requirements engineers, software engineers, and programmers would just create the objects inside the appropriate modules. Also, it would be easy to implement access rights for TOOR modules by which a module would be blocked or have restricted access. At the moment these issues are not under consideration. Again, management and methodological points are not the primary subject of this thesis.

3.10 Discussion

Although we have not presented yet the full traceability mechanisms available in TOOR, the example in the previous section shows how traceability can be improved by the simple use of structuring mechanisms and by the automatic generation of trace information. In whatever phase a system may be, the existence of meaningful modules facilitates the recollection of important information. As seen in the previous example the fact that all concept objects were registered in the same module, DD, made possible a quick inspection of its contents once a problem with attributes was related to undefined concepts.

By careful design of the project structure the user can restrict the creation of objects to meaningful modules and thus avoid registering similar or related information through several different places which for large systems may be difficult to recollect. Module importation makes possible the use of objects wherever they are needed, regardless of the modules where they are registered. Also, the possibility of setting a module structure in advance frees the user from the task of planning the structure of a project. This task may be assigned to an administrator who would define the main modules and how they are interconnected. The analyst and other users would simply create and relate objects in the appropriate modules.

The automatic generation of trace information, by means of automatic generation of relation instances, gives a flexible way of controlling the writing of specifications (if FOOPS is used). More important, this is a way of automatically keeping track of information used in specification texts and it is a powerful mechanism to highlight possibly missing information. If a missing link is detected in a FOOPS specification it may be possible that the specification is not using an agreed naming convention or that it is specifying some concept not related to any previously registered information.

Chapter 4

Using Patterns to Trace Objects

Structuring a project is a valuable aid in tracing the objects in it. The careful determination of module signatures and module interconnection helps to find which module contains a certain class of objects. This reduces the space of search and it also may give valuable hints to where to start a search and what direction to proceed in. But this is not enough: for large projects the number of objects in a module may be so large that little help is got from knowing the module where they are introduced. For example, Davis and Vick [34] report that the development of a ballistic missile defense system, in 1976, had 8,500 requirements. Even in more modest situations with fewer objects, if we add to the picture the relations that exist among objects, it is easy to see that the amount of information may still be large enough to hinder efforts to trace them by simple inspection.

The use of regular expressions for requirements tracing is based on the idea of using patterns to search for actual configurations of objects and relations matching those patterns. In this way we can provide the overall structure of related objects and ask for actual objects fitting the structure. If we want to trace a specification `A` back to documents related to any requirement which is related to `A`, we may express it as a pattern `Doc` *Rel* `Req` *Rel* `A`. This pattern is matched by any object of class `Doc` related (*Rel*) to any object of class `Req` which is related to the object identified by `A`. The result involves only the document and requirement objects related to specification `A` in the way described by the pattern. It leaves out, for instance, documents that are related directly to specification `A`.

Regular expressions are widely used to search texts for strings matching a given pattern [2, 1, 140]. They are very flexible in that one string can match different patterns and one pattern can be matched by different strings. TOOR uses regular expressions in a similar way: the actual configuration of objects and relations is considered as a text and regular expressions are used to retrieve parts of the configuration matching the pattern described by them. The flexibility to express patterns in different ways agrees with our view that requirements are

situated, as are relations involving them. A given relation may have been created by virtue of a certain perception of the related objects that may not be shared by all those who are involved in a project. Two objects may be seen as related in one way by some person and in a different way by another one. Different relations between the same objects may be relevant to different situations. The decision of what relation to consider when tracing an object is dependent on the situation in which the trace is performed, on the purpose of tracing, and on who is tracing the object. TOOR enhances the flexibility of regular expressions by extending the pattern matching procedure by providing different ways of specifying how an object or relation is to be matched.

Tracing with regular expressions is not an isolated feature of TOOR. It is part of an integrated environment for tracing. The user can restrict regular expression tracing to a certain module or group of modules. The user can also use the class specification given by the project specification to characterize the objects to be traced in terms of their properties.

The result of tracing with a regular expression is itself a configuration state: the configuration of objects and relations matching the pattern. This resulting configuration can be used as the basis for the next tracing of objects. Therefore, tracing with regular expressions can be performed in an incremental way, with each result serving to narrow the scope of the next search. Moreover, TOOR presents the resulting configuration in a graphical form which may be browsed and inspected. At any time, the user can switch from one mode of tracing to another.

In this chapter we assume familiarity with regular expressions and automata (good introductions are given in [2, 3]). We explain the different aspects of using regular expression to trace objects in the following way. First we introduce the use of regular expressions to trace objects through an extended example, showing its basic features. Then we give an overview of the tracing mechanism supporting regular expression searches and describe in detail the formal aspects behind it. This is followed by a presentation of advanced features of tracing with regular expressions and by a discussion of its integration with other aspects of TOOR's environment.

4.1 Tracing with Regular Expressions

Regular expressions in TOOR describe patterns of related objects, expressed in terms of object and relation identifiers combined by means of regular operators. For example, the regular expression

```
(req1 | req2) Derive req3
```

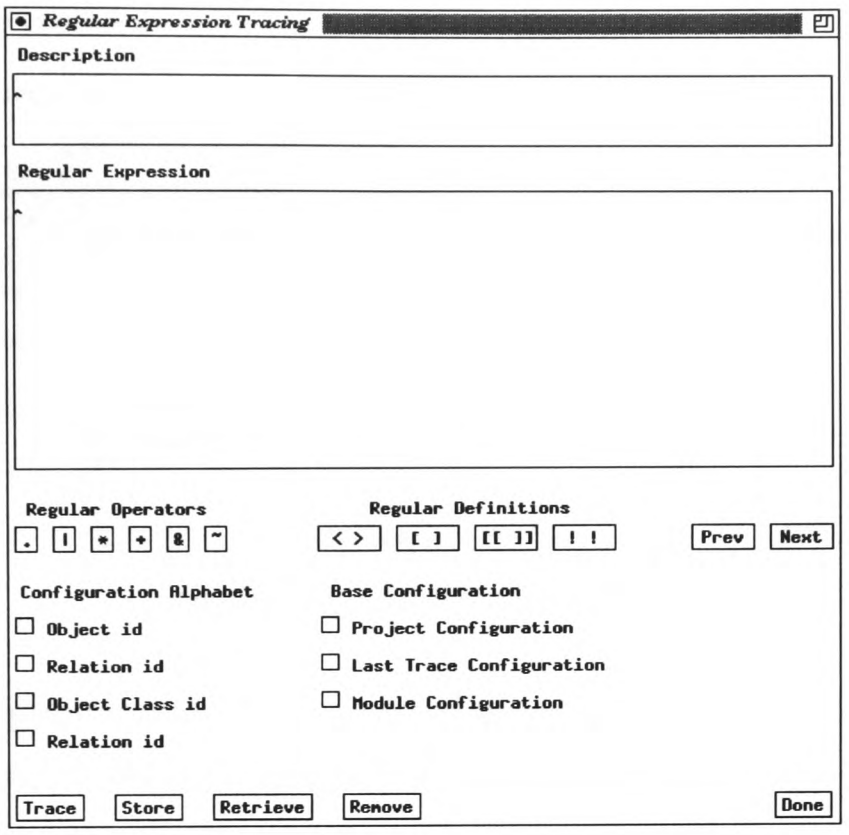


Figure 4.1: Regular Expression Tracing Template

consists of the object identifiers `req1`, `req2`, and `req3`, and the relation class identifier `Derive`. The regular operator | expresses an alternative and concatenation of identifiers is expressed by a space between them. Given a regular expression, TOOR searches its configuration of object and relations for objects related in the way described by the pattern. In the example above an object `req1` related to `req3` by `Derive` would match the pattern.

4.1.1 Using the Tracing Template

Regular expressions are written by the user in the template shown in Figure 4.1. The template has two main fields. The first one is to write a description of the regular expression. It may contain, for example, a natural language description that may be useful if the regular expression is going to be stored and retrieved later on. The description helps to verify the intent of a regular expression if it was written by someone else. The second field is used to write the regular expression itself. The user types in the desired expression but he also may have parts of it automatically written by selecting appropriate buttons. Below this second field there are sets of buttons for the allowed regular operators and regular definitions. Regular definitions are 'names' assigned to regular expressions that can be used in place of the regular expression they define. Therefore, they can be used to make regular expressions shorter and more concise. There are several types of regular definitions in TOOR, differentiated by special delimiters. Regular definitions can be stored for later use. Clicking on a button for regular definitions brings up a menu containing the options *insert*, *create*, *retrieve*, and *delete*.

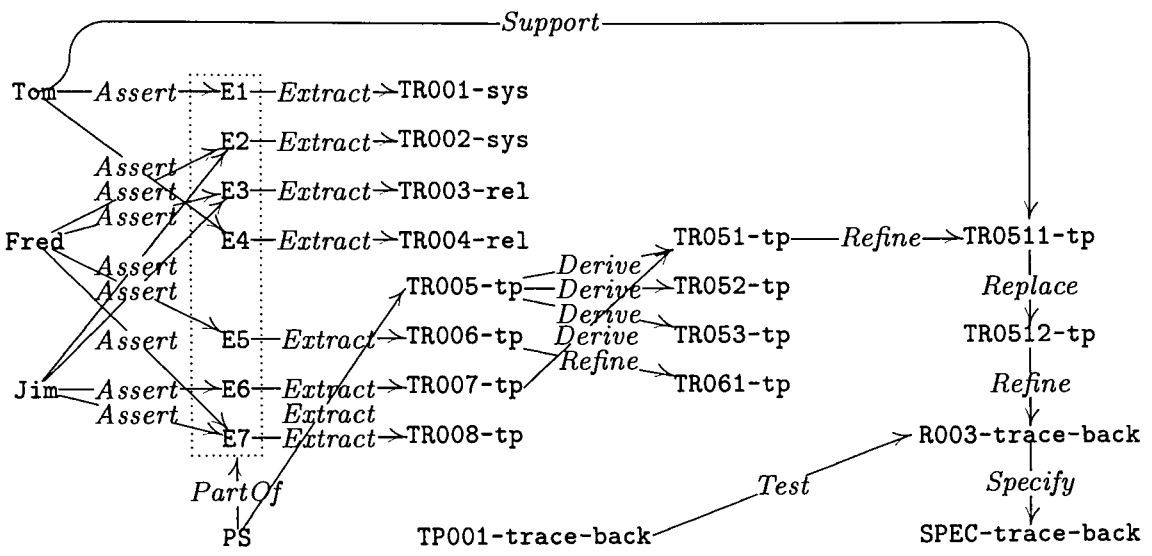


Figure 4.2: Configuration State

Selecting the *insert* option causes the corresponding delimiters to be inserted into the regular expression for the user to write the regular definition in between. The other options are for storing regular definitions and for retrieving and deleting stored ones. Regular expressions themselves can also be stored and retrieved. The buttons labelled *Next* and *Prev* are used to browse stored regular expressions. There are also sets of buttons that may be used to get identifiers for objects and relation classes and to select the base configuration for tracing.

4.1.2 Introductory Example

The use of regular expressions for searching patterns of related objects is best illustrated by means of an example. The example in this section is extracted from [109].

Consider the configuration state of Figure 4.2. It shows a configuration of related objects as a graph where both object and relation instances are nodes, and the edges represent the source and target objects for each relation instance. The relation nodes are shown in italics. The class name is used instead of the identifier for a relation node. The identifier for a particular relation instance is not shown since it is automatically generated by TOOR and it is usually not of interest to the user. For instance, the object `Tom` is related to object `E1` by a relation instance of class `Assert`. Considering all objects as being registered in a single TOOR module `PROJECT` we have

```
tmod PROJECT is
  retain People       : Tom Fred Jim .
  retain Document     : PS .
  retain Statement    : E1 E2 E3 E4 E5 E6 E7 .
  retain TentativeReq : TR001-sys TR002-sys TR003-rel TR004-rel TR005-tp
                        TR006-tp  TR007-tp  TR008-tp  TR051-tp  TR052-tp
                        TR053-tp  TR061-tp  TR0511-tp TR0512-tp .
  retain Requirement  : R003-trace-back .
  retain FOOPSmod     : SPEC-trace-back .
```

```
  retain FOOPStext     : TP001-trace-back .
endt
```

The above module description contains only the declarations for the object instances showing their respective classes. Declarations for relation instances and module signature are not shown. Also, we do not discuss the project specification defining each class. The alphabet used to make regular expressions consists of all object and relation class identifiers. Any additional detail is presented below as necessary.

4.1.3 Searching for an Object

The simplest regular expression we can use consists of a single object identifier. The regular expression

```
TR001-sys
```

is matched by an object having `TR001-sys` as its identifier. In large systems this may be used to find an object given its identifier.

4.1.4 Searching a Sequence of Related Objects

A second use of regular expression is to specify a sequence of related objects. The regular expression

```
Jim Assert E6 Extract TR007-tp
```

is matched by the objects identified by `Jim`, `E6`, and `TR007-tp`, if `Jim` is related to `E6` by an instance of `Assert` and `E6` is related to `TR007-tp` by an instance of `Extract`. This form of regular expression is useful for verifying the existence of, and retrieving, a certain configuration of objects and relations. In our example, searching with the above regular expression results in

$$\texttt{Jim} \text{ —}\textit{Assert}\text{—} \rightarrow \texttt{E6} \text{ —}\textit{Extract}\text{—} \rightarrow \texttt{TR007-tp}$$

as the part of the configuration state matching the given pattern.

4.1.5 Searching Alternative Patterns

The user may also use regular expressions to specify unions or alternative patterns by means of the regular operator `|`. The regular expression

```
Jim (Assert | Extract) E6 (Assert | Extract) TR007-tp
```

is matched by the objects identified by `Jim`, `E6`, and `TR007-tp` provided that `Jim` is related to `E6` by a relation instance of class `Assert` or by a relation instance of class `Extract` and `E6` is also related to `TR007-tp` by either `Assert` or `Extract`. In this example, the result is presented as the following graph:

$$\texttt{Jim} \text{ —}\textit{Assert}\text{ —}\!\!\rightarrow \texttt{E6} \text{ —}\textit{Extract}\text{ —}\!\!\rightarrow \texttt{TR007-tp}$$

This form of regular expression allows the user to search for uncertain configurations of objects and relations. It also may be used to find (retrieve) unconnected parts of a large configuration state. The regular expression

```
(E1 Extract TR001-sys) | (TR051-tp Refine TR0511-tp)
```

results in the following graph

$$\begin{array}{c} \texttt{E1} \text{ —}\textit{Extract}\text{ —}\!\!\rightarrow \texttt{TR001-sys} \\ \texttt{TR051-tp} \text{ —}\textit{Refine}\text{ —}\!\!\rightarrow \texttt{TR0511-tp} \end{array}$$

consisting of two different parts from the original configuration state.

4.1.6 Searching for Repeated Patterns

Iterations of objects (or of configurations of objects and relations) specifying that they may occur zero or more times are expressed by means of the regular operator *. In the regular expression

```
TR005-tp Refine* Derive* TR052-tp
```

the symbol `Refine`, as well as `Derive`, can occur zero or more times. Therefore, the expression is matched by strings of the form

```
TR005-tp Derive TR052-tp,
TR005-tp Refine Derive TR052-tp,
TR005-tp Refine TR052-tp,
TR005-tp TR052-tp, etc.
```

In this example, the result graph is

$$\texttt{TR005-tp} \text{ —}\textit{Derive}\text{ —}\!\!\rightarrow \texttt{TR052-tp}$$

In many situations using only object and relation identifiers is not enough to express a desired pattern. Indeed, the user may not know what identifier he is looking for. Frequently, one wants to trace an object backward or forward to other objects of which only some general information is known. Regular expressions in TOOR are enlarged to allow several other ways of expressing patterns. However, before presenting other types of regular expression searches we describe TOOR's regular expression tracing mechanism and formally define its most important concepts.

4.2 Overview of TOOR's Regular Expression Tracing Mechanism

The use of regular expression for pattern matching presupposes the existence of a text to be searched for strings matching the pattern. Therefore, our first step is to devise a way to represent configurations of objects and relations in a text-like form.

Suppose we have a configuration of objects and relations where requirement `r1` is related to requirement `r2` by a relation `Derive`. We can express this state of affairs by

```
r1 Derive r2.
```

Suppose also that requirement `r2` is related to requirement `r3` by a relation `Refine`. Again, we can express this by

```
r2 Refine r3.
```

We can also express the entire state of affairs by

```
r1 Derive r2 Refine r3.
```

In general, we can represent (part of) a configuration of objects and relations by sequences of object and relation class identifiers. Such sequences are the basis of TOOR's search mechanism. For a given configuration state we consider its set of object and relation class identifiers as an *alphabet*, i.e., a finite set of symbols. The sequences illustrated above are then *strings* from this alphabet, i.e., sequences of symbols. In later sections we give a precise characterization of these notions. For the moment, it suffices to say that for any configuration of objects and relations we can determine the set of all sequences consisting of object names interleaved with relation names that can correctly refer to actual states of affairs, i.e., we can determine the set of all sequences of the form

$$\langle object \rangle \langle Relation \rangle \langle object \rangle \langle Relation \rangle \ \ldots$$

such that if `r1 Derive r2` is a subsequence of it, then in the configuration state there are objects identified by `r1` and `r2` such that `r1` is related by `Derive` to `r2`. This set of sequences forms a *language* (which is defined as any set of strings from a given alphabet). It may also be viewed as a 'text' describing a particular configuration of objects and relations.

A regular expression describes a pattern and denotes a language, i.e, the set of strings matching the pattern. We use regular expressions to express patterns of related objects and to search the 'text' describing a configuration of objects and relations for sequences matching those patterns. The main points of our model can be informally described as follows:

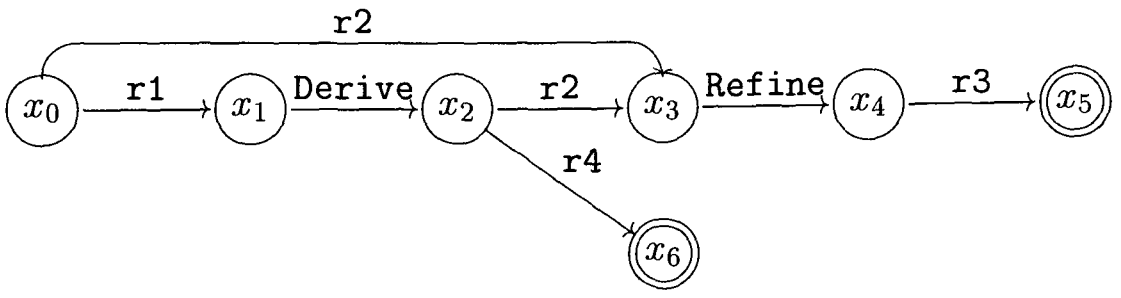


Figure 4.3: Automaton Example (I)

- Configurations of objects and relations are described by sequences of object and relation identifiers. The exact form of these sequences is given by a grammar which also serves to define the language that may be used to refer to related objects, i.e., the set of all such sequences.

- The language referred to above is recognized by an automaton that may be automatically generated from a given configuration state. We call it a *configuration automaton.* A nondeterministic finite automaton (NFA) is defined by *i*) a set of states, *ii*) a set of input symbols, *iii*) an initial state, *iv*) a set of final states, and *v*) a transition function giving the set of states reachable from a given state after receiving an input symbol. A sequence of symbols is recognized by an NFA if starting from the initial state there is a sequence of states, ending on a final state, such that the input symbols making the transition from one state to another spell out the string. An NFA can be represented by a directed labelled graph, in which the nodes are the states and the labelled edges represent the transition function. The final states are conventionally draw with double lines.

Example 52 Consider the set of identifiers {`r1`, `r2`, `r3`, `r4`, `Derive`, `Refine`} as the input alphabet. Then, the NFA represented in Figure 4.3 recognizes the string

`r1 Derive r2 Refine r3`

because the states x_0, x_1, x_2, x_3, x_4, x_5 form a recognizing sequence of states, i.e., x_5 is a final state and the input symbols making the transition from one state to another are exactly `r1`, `Derive`, `r2`, `Refine`, and `r3`, in that order. The string

`r1 Derive r4`

is also recognized by this automaton, in this case the recognizing sequence of states is x_0, x_1, x_2, x_6. On the other hand, the string

`r1 Derive r4 Refine r3`

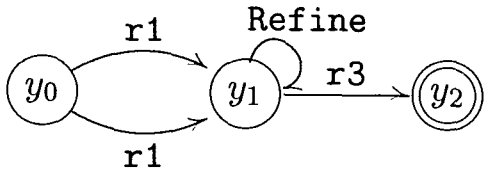


Figure 4.4: Automaton Example (II)

is not recognized by this automaton. □

- The set of all strings referring to a given configuration state, i.e., the language recognized by the corresponding configuration automaton, can be searched using patterns described by regular expressions. For any regular expression R there is an automaton that recognizes the language denoted by R, i.e., it recognizes all strings that match the pattern denoted by R. We call this automaton a *regular expression automaton.*

Example 53 The regular expression

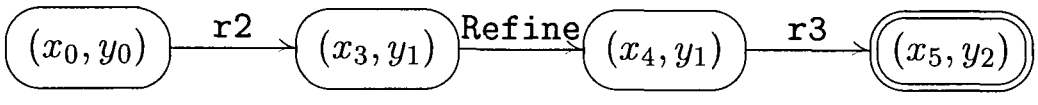
```
(r1 | r2) Refine* r3
```

denotes the language whose strings begins with `r1` or `r2`, ends with `r3`, and contains zero or more occurrences of `Refine` in between. The regular expression automaton represented in Figure 4.4 recognizes any string matching this regular expression. □

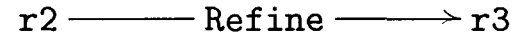
The result of a search using regular expressions is given by the intersection of the configuration automaton and the regular expression automaton. It consists of all strings that conform to the configuration state and match the given regular expression. However, TOOR does not present the result as an automaton since the notation of strings, states, and transitions may not make sense to the user. Automata are just internal mechanisms used by TOOR. Ultimately, the user sees the constituents of a regular expression as object identifiers and not as alphabet symbols. Also, it is not feasible to enumerate all strings in the language of the resulting intersection automaton: their number may be very large and even infinite (in case of loops). The solution adopted is to present the result as the configuration state underlying the intersection automaton, i.e., in terms of a graph whose edges connect related objects (nodes). This graphical form gives a precise idea of the objects found in a regular expression search and of how they are related.

Example 54 Suppose that the automaton of Figure 4.3 is the configuration automaton for a given configuration state S. We can search S for objects matching the pattern described by the regular expression of Example 53. This amounts to find the strings common to both

automata: the configuration automaton of Figure 4.3 and the regular expression automaton of Figure 4.4. The intersection automaton is given by



and the result of tracing, after some transformation is given by



This only illustrates the steps followed in tracing an object. The details of how to build the intersection automaton and how the resulting graph is generated are given in Section 4.5. □

The tracing procedure can be summarized as follows: for a configuration state S, given a regular expression R over the alphabet of object and relation class identifiers we proceed in the following way:

1) Construct the configuration automaton A_S.

2) Construct the regular expression automaton A_R.

3) Construct the automaton $A_{S \cap R}$, intersecting A_S and A_R.

4) Present the configuration state underlying $A_{S \cap R}$ as the result of the regular expression search.

These steps are formally described in the following sections.

4.3 Referring to Objects and Relations

Our alphabet consists of object and relation class identifiers. We use the following notation to denote these sets of identifiers.

The set of object classes and relation classes identifiers are denoted by $\mathcal{I}_{oc}$ and $\mathcal{I}_{rc}$, respectively. We have that $\mathcal{I}_{oc} \cap \mathcal{I}_{rc} = \emptyset$. The set of all class identifiers is denoted by $\mathcal{I}_{class} = \mathcal{I}_{oc} \cup \mathcal{I}_{rc}$. The set of object and relation identifiers are denoted by $\mathcal{I}_o$ and $\mathcal{I}_r$, respectively. We have that $\mathcal{I}_o \cap \mathcal{I}_r = \emptyset$. The set of all object identifiers is denoted by $\mathcal{I}_{obj} = \mathcal{I}_o \cup \mathcal{I}_r$, because relations are also objects (and can be related to other objects).

A configuration of objects and relations determines, at any time, the sets of object and class identifiers, i.e., the set of identifiers referring to the actual objects registered in TOOR and to the existing relation classes in the project's signature.

Definition 55 For a configuration S of objects and relations, the corresponding **configuration alphabet** I is given by $\mathcal{I}_{obj} \cup \mathcal{I}_{rc}$ □

The strings we use to talk about objects and their relationships are built out of the configuration alphabet. They consist of object identifiers interleaved with relation class identifiers. The intuitive meaning of a string $o_1 r_1 o_2$ is to say that object o_1 is related to object o_2 by relation r_1. These strings are precisely defined by

Definition 56 A **configuration grammar** is defined as $G = (I_N, I, X, F)$, where $I = \mathcal{I}_{obj} \cup \mathcal{I}_{rc}$ is the alphabet, $I_N = \{X\}$ is the set of non-terminal symbols, X is the initial symbol, and

$$\begin{array}{lcl} X & \rightarrow & o_i \\ X & \rightarrow & o_i r_j X \end{array}$$

are the production rules denoted by F. The symbol o_i, for object identifier, ranges over the elements of $\mathcal{I}_{obj}$, i.e., there is one production rule containing o_i for each symbol o_i in $\mathcal{I}_{obj}$. Similarly, r_j ranges over the elements of $\mathcal{I}_{rc}$. □

This grammar specifies the valid strings to be used when referring to objects and relations. They are constructed in the following way: starting from the initial symbol, replace the right-hand side of some production rule for the non-terminal symbol at the left-hand side of that production rule, and finish when the string has only terminal symbols in it. For example, the string $o_1 r_1 o_2 r_2 o_3$ is constructed by performing the substitutions below.

$$X \rightarrow o_1 r_1 X \rightarrow o_1 r_1 o_2 r_2 X \rightarrow o_1 r_1 o_2 r_2 o_3$$

Observe that identifiers are taken as symbols of our alphabet whilst they are themselves strings of characters. This is not a problem since they are unique and we can clearly distinguish them by placing a special symbol between two consecutive identifiers. In our example we use spaces to separate distinct symbols.

We should note that the set of relation class identifiers is part of the project signature and changes every time the project specification is modified such that new classes are added or old ones are removed. The set of object identifiers changes every time an object is created or removed.

4.3.1 Referring to Configuration States

The configuration grammar determines the valid strings of object and relation identifiers. The next step is to precisely describe what the *correct* strings are in the sense that if in a

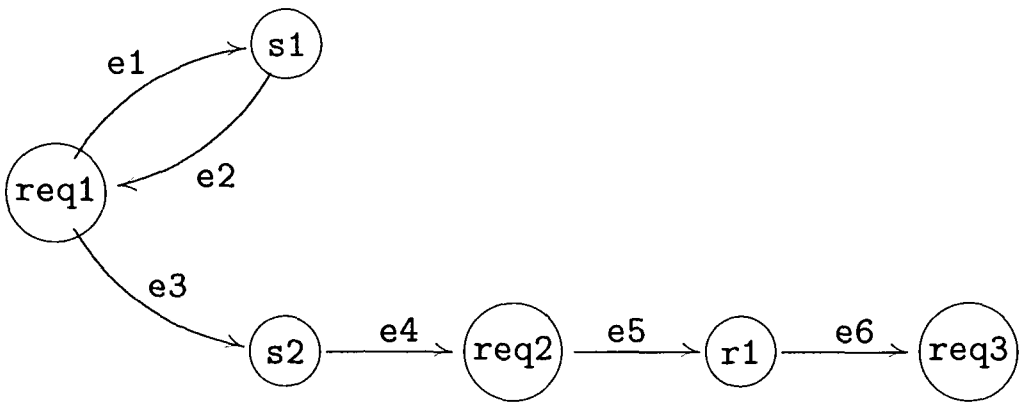


Figure 4.5: Configuration State

string an identifier a is related to b by R, then there is an actual object identified by a related by a relation instance of class R to an actual object identified by b.

The configuration of objects and relations in TOOR can be precisely described by a graph, with object and relation instances as nodes, arrows representing the source and target of relations, and labels denoting the class names of objects and relations.

Definition 57 A **configuration state** S is a directed labelled graph (N, E, L, v_0, v_1, v_l) where N is the set of nodes, E is the set of edges, L is the set of labels, v_0 and v_1 are functions from E to N giving to each edge its initial and terminal nodes, and v_l is a function from N to L giving to each node a label, such that

1) the set N of nodes consists of the objects and relations, i.e., $N = \mathcal{I}_{obj} = \mathcal{I}_O \cup \mathcal{I}_r$;

2) for each relation object $r \in \mathcal{I}_r$, with objects o_1 and o_2 as source and target respectively, there are edges s_r and t_r in E such that $v_0(s_r) = o_1$, $v_1(s_r) = v_0(t_r) = r$, and $v_1(t_r) = o_2$; and

3) the set L of labels consists of the class names, i.e., $L = \mathcal{I}_{class} = \mathcal{I}_{oc} \cup \mathcal{I}_{rc}$. For each object o of class C the label of the corresponding node is the name of its class, i.e., $v_l(o) = C$.

□

Example 58 Consider the situation where objects req1, req2, req3 are registered in a project together with relation instances r1, s1, and s2. The class of each object and the source and target attributes for each relation instance are shown in the table below.

object id	class	attributes
req1	Req	
req2	Req	
req3	Req	
r1	Derive	source: req2 target: req3
s1	Refine	source: req1 target: req1
s2	Refine	source: req1 target: req2

The sets of object identifiers and class names are given by

$$\begin{aligned} \mathcal{I}_O &= \{\texttt{req1}, \texttt{req2}, \texttt{req3}\} \\ \mathcal{I}_R &= \{\texttt{r1}, \texttt{s1}, \texttt{s2}\} \\ \mathcal{I}_{OC} &= \{\texttt{Req}\} \\ \mathcal{I}_{RC} &= \{\texttt{Derive}, \texttt{Refine}\} \end{aligned}$$

the configuration alphabet is given by $V = \{\texttt{req1}, \texttt{req2}, \texttt{req3}, \texttt{r1}, \texttt{s1}, \texttt{s2}, \texttt{Derive}, \texttt{Refine}\}$, and among the valid strings that may be generated by the configuration grammar are

```
req1 Refine req2
req1 Refine req2 Derive req3
req1 Refine req1 Refine req2 Derive req3
req1 Derive req2
req2 Derive req2 Derive req2.
```

Only the first three correctly refer to the actual configuration of objects and relations according to the intuitive meaning we give to them. The fourth string (intuitively) states that `req1` is related to `req2` by relation `Derive`. This does not correspond to the way objects are actually related. The same can be said of the fifth string.

The corresponding configuration state is given by the graph of Figure 4.5. The functions v_0 and v_1 are defined in the obvious way and the label function is defined by

$$\begin{aligned} v_l(\texttt{req1}) = v_l(\texttt{req2}) &= v_l(\texttt{req3}) = \texttt{Req} \\ v_l(\texttt{r1}) &= \texttt{Derive} \\ v_l(\texttt{s1}) = v_l(\texttt{s2}) &= \texttt{Refine} \end{aligned}$$

□

The use of a configuration state to represent the objects and their relationships is accurate in the sense that, at any point in time, there is a one to one correspondence between the objects registered in a project and the nodes in the corresponding configuration state. It is easy to see, from the definition of a configuration state S, that

1) For every object o of class C in TOOR, there is a node o in S such that $v_l(o) = C$. Conversely, for every node o in S, with $v_l(o) = C$, there is an object o of class C in TOOR.

2) For every relation instance r of class R with objects x of class A and y of class B as its source and target objects, respectively, there is a node r and edges s_r and t_r in S such that

   1. $v_0(s_r) = x$, $v_1(s_r) = v_0(t_r) = r$, $v_1(t_r) = y$.
   2. $v_l(r) = R$, $v_l(x) = A$ and $v_l(y) = B$.

   Conversely, for every node r and edges s_r and t_r in S, such that

   1. $v_1(s_r) = v_0(t_r) = r$, $v_0(s_r) = x$, $v_1(t_r) = y$.
   2. $v_l(r) = R$, $v_l(x) = A$ and $v_l(y) = B$.

   there is a relation instance r of class R with objects x of class A and y of class B as its source and target objects, respectively.

4.3.2 String Conformance

As shown in Example 58 not all strings generated by a configuration grammar can correctly refer to the corresponding configuration state. We introduce the notion of string conformance to define those strings that correctly refer to a given configuration state.

A string $w = s_0 \dots s_n$ conforms to a configuration of objects and relations if for each object identifier s_i in w there is an object o identified by s_i, and for each relation class identifier s_j there is a relation instance r whose class is identified by s_j and such that the source object of r is identified by s_{j-1} and the target object of r is identified by s_{j+1}.

Example 59 The string `req1 Refine req2` conforms to a configuration state S if there are in S objects identified by `req1`, `req2`, and a relation instance `rel` whose class is identified by `Refine` and whose source and target objects are `req1` and `req2`, respectively. □

The notion of string conformance can be precisely stated using the definition of a configuration state.

Definition 60 Giving a configuration state $S = \{N, E, L, v_0, v_1, v_l\}$, a string $s_0 s_1 \dots s_n$ generated by G_S **conforms** to S if there are nodes $o_0, \dots, o_n \in N$ such that

(1) $s_0 = o_0$

(2) $s_i = o_i$, $v_0(e) = o_{i-1}, v_1(e) = o_i$,
for some $e \in E$; for $0 = i \bmod 2$, $i = 2, \ldots, n$; and

(3) $s_i = v_l(o_i), v_0(e) = o_{i-1}, v_1(e) = o_i$,
for some $e \in E$; for $1 = i \bmod 2$, $i = 1, \ldots, n-1$.

□

Example 61 Consider the configuration state of Figure 4.5 described in Example 58. The string `req2 Refine req2 Derive req3` (which is decomposed as $s_0 =$ `req2`, $s_1 =$ `Refine`, $s_2 =$ `req2`, $s_3 =$ `Derive`, and $s_4 =$ `req3`) does not conform to this configuration state because there are no nodes o_0, o_1, o_2, o_3, o_4, satisfying the conditions of Definition 60, i.e.,

$$
\begin{array}{ll}
s_0 = \texttt{req2} = o_0 & \\
s_1 = \texttt{Refine} = v_l(o_1), v_0(e) = \texttt{req2} = o_0, v_1(e) = o_1 & \text{, for some } e \in E \\
s_2 = \texttt{req2} = o_2, v_0(e) = o_1, v_1(e) = \texttt{req2} = o_2 & \text{, for some } e \in E \\
s_3 = \texttt{Derive} = v_l(o_3), v_0(e) = \texttt{req2} = o_2, v_1(e) = o_3 & \text{, for some } e \in E \\
s_4 = \texttt{req3} = o_4, v_0(e) = o_3, v_1(e) = \texttt{req3} = o_4 & \text{, for some } e \in E
\end{array}
$$

□

In TOOR it is possible for a relation class to be a subclass of another relation class. Therefore, it is possible for a relation instance of a class `B` to be referred to as an object of class `A` provided that `B` is a subclass of `A`. String conformance takes account of subclass relationships. In fact the third condition of Definition 60 should have been written as

(3) $s_i =_{\prec} v_l(o_i), v_0(e) = o_{i-1}, v_1(e) = o_i$,
for some $e \in E$; for $1 = i \bmod 2$, $i = 1, \ldots, n-1$.

where $s_i =_{\prec} v_l(o_i)$ indicates that the label of the node o_i should be equal to the identifier s_i or it should identify a subclass of the class identified by s_i.

Example 62 Consider a situation where the relation class `Extract`, intended to relate objects of class `Document` to objects of class `Requirement`, has the subclasses `AutomaticExtr` and `ManualExtr`. In a configuration state where document `docA` is related to requirement `req1` by an instance of `ManualExtr`, we have that both strings

`docA ManualExtr req1` and `docA Extract req1`

conform to the configuration state. □

4.4 Viewing Objects and Relations as a Text

Tracing objects using regular expressions involves searching the configuration state for strings matching the patterns of objects and relations described by a particular regular expression. However, there are many different strings that may be used to refer to a given configuration state. We define how these strings can be generated (or recognized) in order for them to be searched and used in a pattern matching procedure.

Having established what strings can correctly refer to a given configuration state we are now in a position to define the set of all such strings. We begin by describing an automaton that recognizes these strings.

Definition 63 Given a configuration state $S = (N, E, L, v_0, v_1, v_l)$, with $N = \mathcal{I}_{obj}$ and $L = \mathcal{I}_{class}$, the associated **configuration automaton** is defined as $A_S = (X, I, \delta, x_I, F)$, where

- $X = X_{obj} \dot{\cup} X_r \dot{\cup} \{x_I\}$ is the set of states. There is one state $x_i \in X_{obj}$ for each node $o_i \in \mathcal{I}_{obj}$, one state $x_r \in X_r$ for each node $o_r \in \mathcal{I}_r$, plus one additional state x_I,
- x_I is the initial state,
- $F = X_{obj}$ is the set of final states,
- $I = \mathcal{I}_{obj} \cup \mathcal{I}_{rc}$ is the input alphabet consisting of object identifiers and relation class identifiers, and
- $\delta : X \times I \to 2^X$ is a function defined as

$$\delta(x_i, j) = \begin{cases} \{x_j \in X_{obj} \mid o_j \in \mathcal{I}_{obj}\} & \text{if } x_i = x_I, \\ \{x_r \in X_r \mid v_l(o_r) = j, v_0(e) = o_i, v_1(e) = o_r, \text{ for some } e \in E\} & \text{if } x_i \in X_{obj}, \\ \{x_j \in X_{obj} \mid v_0(e) = o_i, v_1(e) = o_j, \text{ for some } e \in E\} & \text{if } x_i \in X_r. \end{cases}$$

where, for any $x_w \in X_{obj}$ its corresponding node is denoted by o_w. □

Example 64 Consider the configuration state $S = (N, E, L, v_0, v_1, v_l)$ of Figure 4.5. We describe its sets of nodes and labels again for ease of reference: $N = \mathcal{I}_{obj} = \mathcal{I}_o \cup \mathcal{I}_r$ and $L = \mathcal{I}_{class} = \mathcal{I}_{oc} \cup \mathcal{I}_{rc}$, where

$$\begin{aligned} \mathcal{I}_o &= \{\texttt{req1}, \texttt{req2}, \texttt{req3}\} \\ \mathcal{I}_r &= \{\texttt{r1}, \texttt{s1}, \texttt{s2}\} \\ \mathcal{I}_{oc} &= \{\texttt{Req}\} \\ \mathcal{I}_{rc} &= \{\texttt{Derive}, \texttt{Refine}\} \end{aligned}$$

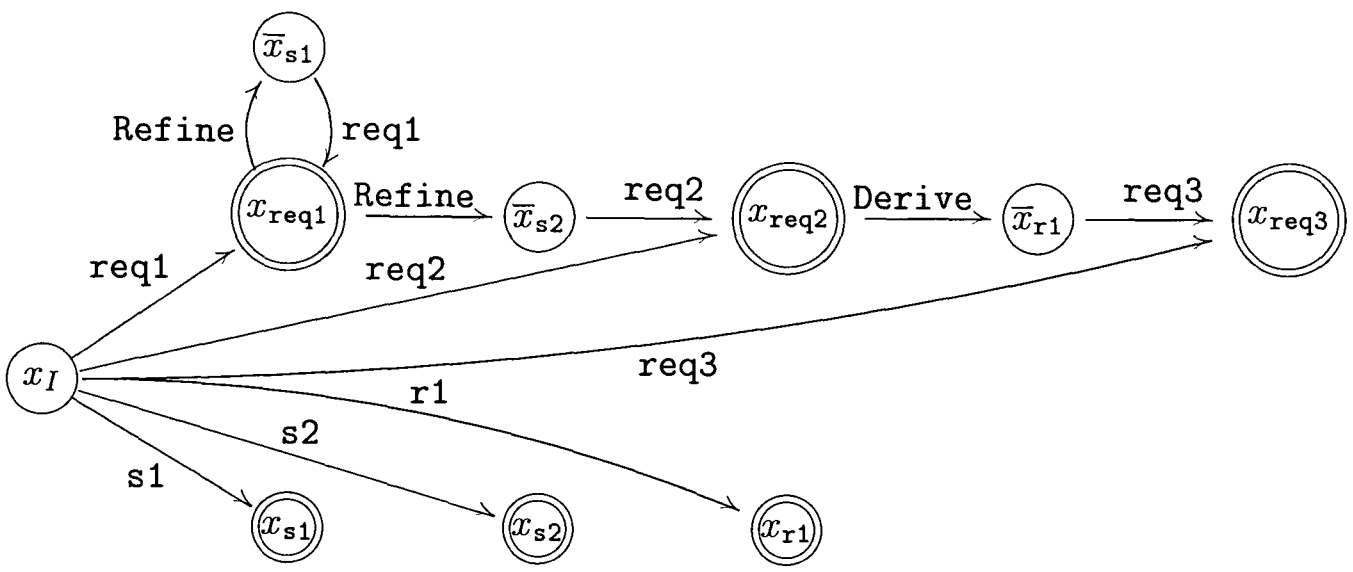


Figure 4.6: Configuration Automaton

The corresponding configuration automaton $A_S = (X_S, I, \delta_S, x_I, F_S)$ is represented by the graph of Figure 4.6. It has the following states and input alphabet:

$$\begin{aligned} I &= \mathcal{I}_{obj} \cup \mathcal{I}_{rc} = \{\mathtt{req1}, \mathtt{req2}, \mathtt{req3}, \mathtt{s1}, \mathtt{s2}, \mathtt{r1}, \mathtt{Derive}, \mathtt{Refine}\}. \\ X_S &= X_{obj} \dot{\cup} X_r \dot{\cup} \{x_I\}. \\ X_{obj} &= \{x_{\mathrm{req1}}, x_{\mathrm{req2}}, x_{\mathrm{req3}}, x_{\mathrm{s1}}, x_{\mathrm{s2}}, x_{\mathrm{r1}}\}. \\ X_r &= \{\overline{x}_{\mathrm{s1}}, \overline{x}_{\mathrm{s2}}, \overline{x}_{\mathrm{r1}}\}. \end{aligned}$$

This automaton recognizes, among other strings, the strings

`req1 Refine req2 Derive req3`,

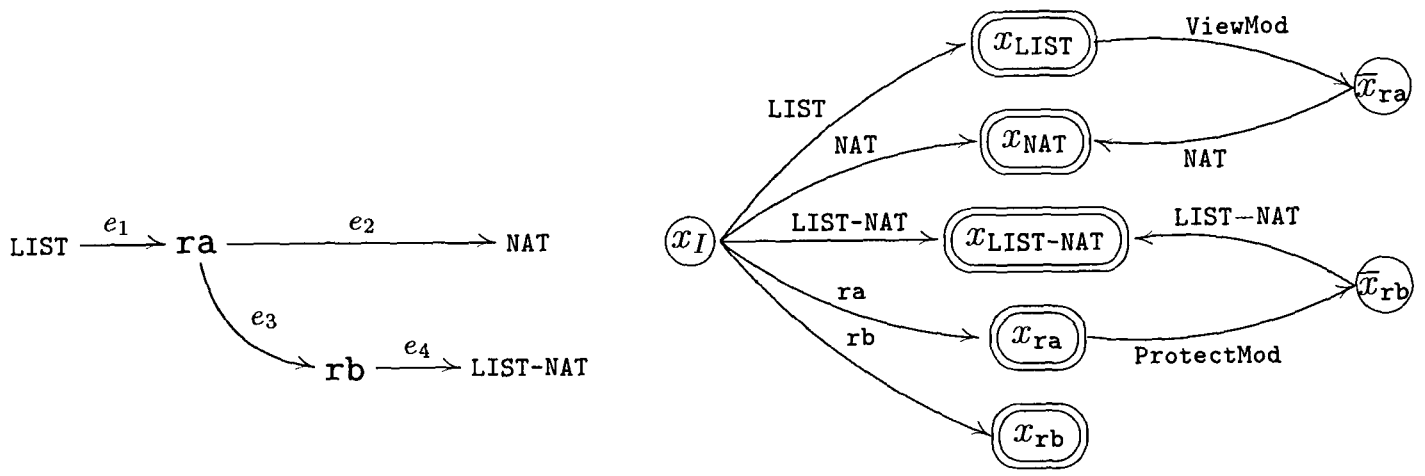
`req1 Refine req1`, and

`req1`.

□

Observe that we overline the states in X_r. This is a notational convenience. For each relation instance there are two states with only one corresponding to a final state. We denote both states by the same symbol with the overlined one not being a final state. The existence of two states for each relation instance is a simple way to represent relations of relations and at the same time to avoid the recognition of strings having two consecutive class identifiers or ending with a relation class identifier.

Example 65 In Example 51, in Chapter 3, we presented a situation where relations of relations occur naturally in TOOR. If a FOOPS module containing an instantiation of a generic module is registered as an object, then one of the automatic relations generated by TOOR involves the relation instance corresponding to the module instantiation. This situation is reproduced with the FOOPS module



(A) (B)

Figure 4.7: (A) Configuration State (B) Configuration Automaton

```
fmod LIST-NAT is
  protecting LIST[NAT] .
endf
```

being registered as an object in a TOOR module SYS-SPEC. The module SYS-SPEC, after the automatic generation of relations, becomes

```
tmod SYS-SPEC is
  retain FOOPSmod   : LIST NAT LIST-NAT .
  retain ViewMod    : (LIST,NAT) .
  retain ProtectMod : ((LIST,NAT), LIST-NAT) .
endt
```

Suppose that these are the only objects and relations registered in TOOR. Then, the sets of identifiers are given by

$$\mathcal{I}_O = \{\texttt{LIST}, \texttt{NAT}, \texttt{LIST-NAT}\}$$
$$\mathcal{I}_r = \{\texttt{ra}, \texttt{rb}\}$$
$$\mathcal{I}_{OC} = \{\texttt{FOOPSmod}\}$$
$$\mathcal{I}_{rC} = \{\texttt{ViewMod}, \texttt{ProtectMod}\}$$

where ra is the identifier of the relation instance of class ViewMod relating LIST to NAT and rb is the identifier of the relation instance of class ProtectMod relating ra to LIST-NAT. The configuration state is shown in Figure 4.7(A) with its nodes labelled by

$$v_l(\texttt{LIST}) = v_l(\texttt{NAT}) = v_l(\texttt{LIST-NAT}) = \texttt{FOOPSmod}$$
$$v_l(\texttt{ra}) = \texttt{ViewMod}$$
$$v_l(\texttt{rb}) = \texttt{ProtectMod}$$

The corresponding automaton is shown in Figure 4.7(B) with its alphabet and states given by

$$I = \{\texttt{LIST}, \texttt{NAT}, \texttt{LIST-NAT}, \texttt{ViewMod}, \texttt{ProtectMod}\}$$

$$X_{obj} = \{x_{\texttt{NAT}}, x_{\texttt{LIST}}, x_{\texttt{LIST-NAT}}, x_{\texttt{ra}}, x_{\texttt{rb}}\}$$
$$X_r = \{\overline{x}_{\texttt{ra}}, \overline{x}_{\texttt{rb}}\}$$

Observe that from a state in X_{obj} a transition can only occur to a state in X_r, upon the receiving of a relation class identifier, and that from a state in X_r a transition can only occur to a state in X_{obj}, upon the receiving of an object identifier. Therefore, it is not possible for a string containing two adjacent relation class identifiers to be recognized by a configuration automaton. □

The following proposition ascertains the appropriateness of configuration automata to talk about configurations of objects and relations.

Proposition 66 A string $w = s_0 s_1 \ldots s_n$ conforms to a configuration state S if and only if w can be recognized by the corresponding configuration automaton A_S.

Proof: The proof is presented in Appendix A. □

The language accepted by an automaton A is the set of all strings recognized by A. Therefore, by Proposition 66 the language accepted by a configuration automaton A_S consists of all strings one can use to refer to objects and relations in a configuration state S.

4.5 Finding the Matching Strings

A regular expression R denotes the language consisting of all strings that match the pattern described by R. Given a regular expression R, there is an ε-free nondeterministic finite automaton (NFA) A_R which accepts the language denoted by R [3, 11, 19], which we call *regular expression automaton.* The general idea in using regular expressions is to use the configuration automaton A_S and the regular expression automaton A_R to find the strings that are in both languages. We do this by constructing the intersection automaton $A_{S \cap R}$. The intersection automaton is used to find the configuration state containing the objects and relations matching the pattern described by R. This configuration state is presented as the result of the search.

Given two ε-free NFA, their intersection automaton is constructed in such a way that it recognizes strings that are in the language of both automata. The set of states consists of the pair of initial states plus the pairs of states, one from each automaton, which are reachable after input of the same symbol. The set of final states consists of those pairs whose elements are final states of their respective automata. There is a transition from a state (x, y) to a state (x', y') under a symbol s if there are transitions from x to x' and from y to y' under the same symbol s.

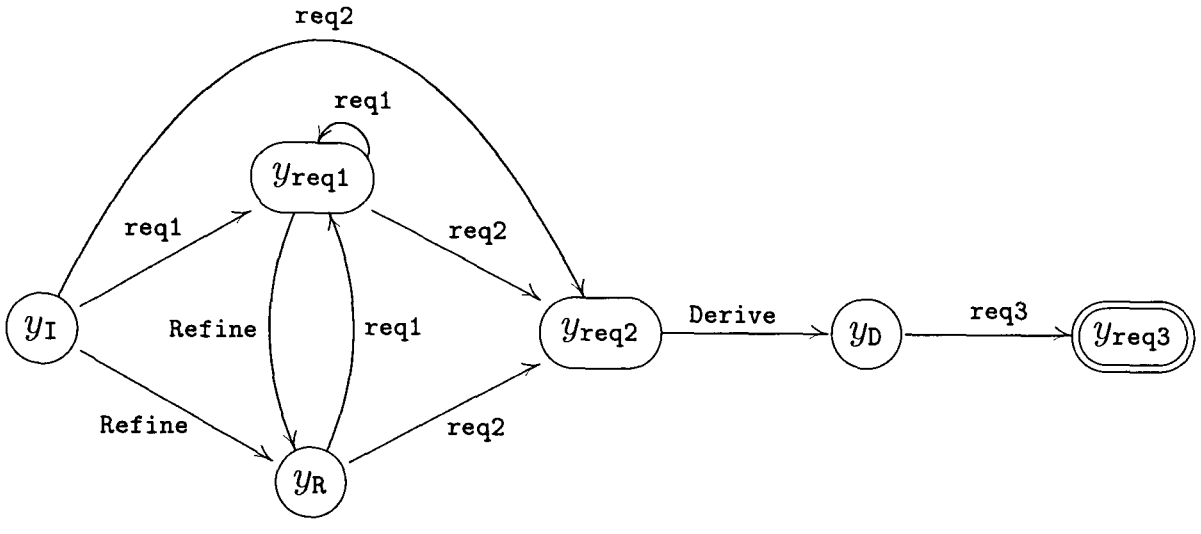


Figure 4.8: Regular Expression Automaton

Definition 67 Let $A_S = (X_S, I_S, \delta_S, x_I, F_S)$ and $A_R = (X_R, I_R, \delta_R, y_I, F_R)$ be two ε-free NFA. We define the **intersection automata** $A_{S\cap R} = (Z, I_{S\cap R}, \delta_{S\cap R}, z_I, F_{S\cap R})$ as

- $Z = \{(x, y) \mid x \in \delta_S(x', a)$ and $y \in \delta_R(y', a)$ for some $x' \in X_S, y' \in X_R, a \in I_{S\cap R}\} \cup \{(x_I, y_I)\}$.
- $I_{S\cap R} = I_S \cap I_R$.
- $z_I = (x_I, y_I)$ is the initial state.
- $\delta_{S\cap R} : Z \times I_{S\cap R} \to 2^Z$ is defined as

$$\delta_{S\cap R}((x, y), a) = \{(x', y') \mid x' \in \delta_S(x, a) \text{ and } y' \in \delta_R(y, a)\}.$$

- $F_{S\cap R} = \{(x, y) \mid x \in F_S$ and $y \in F_R\}$.

□

The above definition requires two NFA but it is still valid for Deterministic Finite Automata (DFA). First, a DFA has no ε transitions. Second, the fact that for any given input symbol a there is at most one transition under a from any state of a DFA, just reduces the number of states in the resulting intersection automaton.

Example 68 Suppose that the regular expression

```
(req1 | Refine)* req2 Derive req3
```

is used to trace objects in the configuration state described in Example 64. The configuration automaton A_S, corresponding to this configuration state, is shown in Figure 4.6. The regular expression automaton A_R, corresponding to the regular expression above, is shown in Figure 4.8. The table below shows for both automata which states are reached after input of the same symbol.

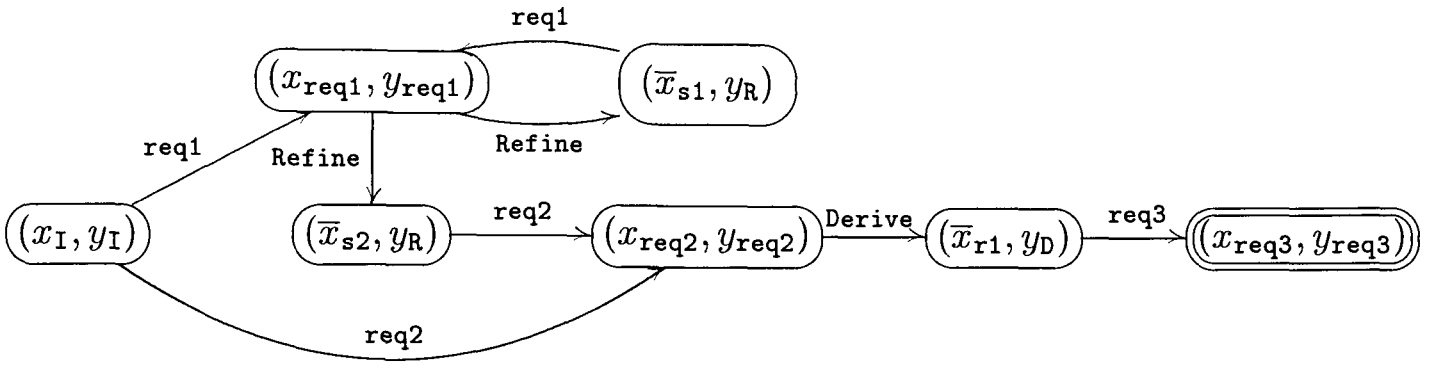


Figure 4.9: Intersection Automaton

Input symbol	States reached after input symbol	
	in A_S	in A_R
req1	X_{req1}	Y_{req1}
req2	X_{req2}	Y_{req2}
req3	X_{req3}	Y_{req3}
Refine	$\overline{X}_{s_1}$, $\overline{X}_{s_2}$	Y_R
Derive	$\overline{X}_{r_1}$	Y_D

The states of the intersection automaton are formed by pairing the states shown in the table above. The initial state is given by (x_I, y_I) and there is only one final state: $(X_{\text{req3}}, Y_{\text{req3}})$. The transitions from one state to another are indicated in the diagram of Figure 4.9. □

Intersection automata may have *loose* states, i.e., states that are not reachable from the initial state. This in no way affects the results presented here because those states are not part of any recognizing sequence of states. Nevertheless, we will consider that no such states exist: they are deleted from the resulting automata.

The intersection automaton determines which strings are in both automata.

Proposition 69 Let A_S and A_R be NFA and $A_{S\cap R}$ their intersection automaton. We have that a string w is recognized by $A_{S\cap R}$ if and only if w is recognized by A_S and A_R. □

4.5.1 Presenting Regular Expression Tracing Results

Given a configuration state S and a regular expression R we have that the language recognized by the automaton intersecting A_S and A_R consists of all strings that conforms to S and matches the pattern described by R. However, as explained before, the result of a regular expression search is not given in terms of the intersection automaton.

To extract the configuration state underlying an intersection automaton we should observe that every intersection automaton contains part of both intersected automata recognizing a

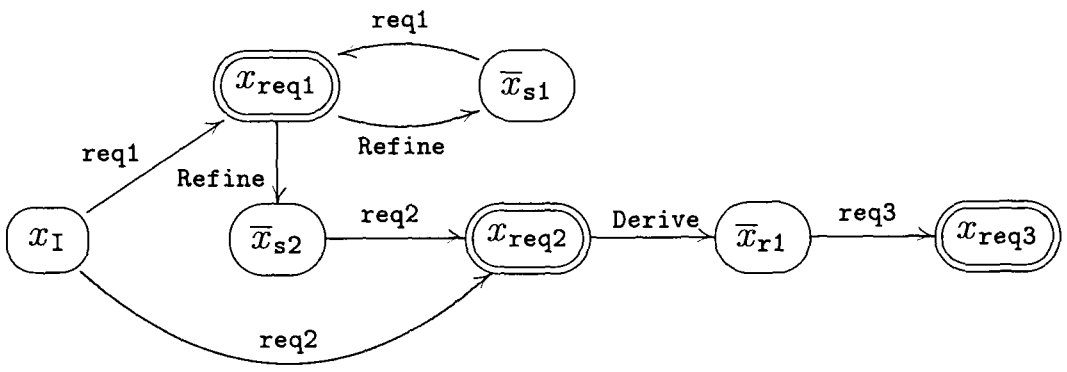


Figure 4.10: Left Sub-Automaton

common language. Thus, every state in an intersection automaton $A_{S\cap R}$ is a pair whose first element comes from A_S and every transition from (x, y) to (x', y') corresponds to a transition from x to x' in A_S. Therefore, given an intersection automaton $A_{S\cap R}$ we can get the nodes and transitions from A_S contributing to its construction. We define the left sub-automaton $A_{S\cap R|_S}$ of an intersection automaton as

Definition 70 For any intersection automaton $A_{S\cap R} = (Z, I, \delta_{S\cap R}, z_I, F_{S\cap R})$ we define its **left sub-automaton** as $A_{S\cap R|_S} = (X'_S, I, \delta'_S, x'_I, F'_S)$, where

- $X'_S = \{x \mid (x, y) \in Z\}$,
- $x'_I = x_I$ for $z_I = (x_I, y_I)$,
- $\delta'_S(x, a) = \{x' \mid (x', y') \in \delta_{S\cap R}((x, y), a)$ for some $(x, y) \in Z\}$, and
- $F'_S = \{x \mid (x, y) \in F_{S\cap R}\}$.

□

The left sub-automaton $A_{S\cap R|_S}$, in our case, is a sub-automaton of A_S. To get the configuration state underlying a configuration automaton we should just get the nodes corresponding to the states of the configuration automaton and the edges corresponding to its transitions. We should recall from the definition of a configuration automaton (Definition 63) that every state x_{o_i} (except for the initial state) in a configuration automaton corresponds to a node o_i in the configuration state and that every transition from a state x_{o_i} to a state x_{o_j} correspond to an edge from node o_i to node o_j. When generating a configuration automaton from a configuration state, TOOR keeps track of the nodes and edges corresponding to each state and transition. Therefore, given a configuration automaton (or part of it) the reconstruction of the corresponding configuration state (or part of it) is immediate.

Example 71 The left sub-automaton $A_{S\cap R|_S} = (X'_S, I, \delta'_S, x'_I, F'_S)$ of $A_{S\cap R}$ of Example 68 is given in Figure 4.10. Observe that every node and every transition of $A_{S\cap R|_S}$ is also a

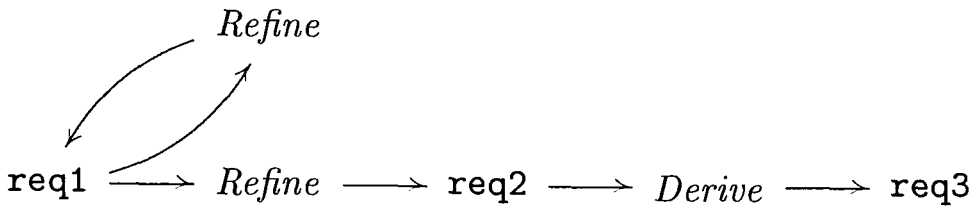


Figure 4.11: Resulting Configuration State

node and a transition of A_S. The resulting configuration state is given in Figure 4.11. Class names are used for relation instances instead of their identifiers. □

4.6 Extending the Regular Expression Language

We now return to the example described in Section 4.1.2 in order to show other features of tracing with regular expressions. All examples in this section refer to the configuration state of Figure 4.2.

We extend TOOR's regular expression language by making it possible to define symbols using *regular definitions* over an extended alphabet. Regular definitions are themselves regular expressions. In fact, they may be regarded as names, given to some regular expressions, that may be used to shorten long regular expressions and make them more readable.

Example 72 A classical example of regular definition for compilers is the definition of an identifier as a sequence of letters and digits beginning with a letter. Instead of expressing an identifier as

```
(A | B | ...| Z | a | b | ...| z)((A | B |...| Z |
                                  a | b | ...| z) | (0 | 1 | ...| 9))*
```

the following definitions may be provided

```
letter → (A | B | ...| Z | a | b | ...| z)
 digit → (0 | 1 | ...| 9)
    id → letter(letter | digit)*
```

and **id** may be used whenever an identifier has to be expressed. □

We should recall that object identifiers are taken as symbols of the configuration alphabet and their occurrences in a regular expression are matched by objects having identical identifiers. However, object identifiers are themselves made out of letters, digits, and special symbols like hyphen (-) and underscore (_). We call the set of symbols used to make identifiers the *identifier alphabet*. Therefore, in TOOR regular definitions are regular expressions

over a different alphabet: the identifier alphabet. They are used inside regular expressions for requirements tracing to express that the symbols they define are to be matched according to the given definition.

4.6.1 Regular Definition for Object Identifier

Regular definitions for object identifiers are used in regular expressions for requirements tracing surrounded by angle brackets (<,>). Any symbol of the form `<defid>`, where `defid` is a regular expression over the identifier alphabet, is to be matched by any object whose identifier matches `defid`.

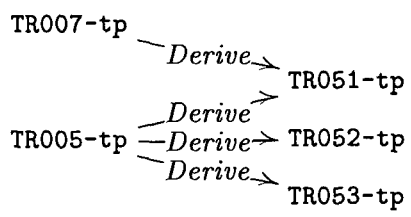
Example 73 The regular definitions

```
digit  →  (0 | 1 | ...| 9)
 treq  →  TR(digit)*-tp
```

express that `treq` is to be matched by any sequence of letters and digits beginning with `TR`, followed by zero or more digits, and ending with `-tp`. It may be used in a regular expression for requirements tracing to specify objects whose identifiers obey some general pattern. The regular expression

```
<treq> Derive <treq>
```

expresses a pattern to be matched by any two objects whose identifiers match `TR(digit)*-tp` and that are related by an instance of `Derive`. In our example, tracing the configuration state of Figure 4.2 with this regular expression produces



as the resulting graph. Observe that there are two ways of using a regular definition. If the definition is registered in TOOR its name may be used, otherwise it should be written in full using the symbols from the identifier alphabet. In both cases, the regular definition should be surrounded by angle brackets. □

4.6.2 Regular Definition for Classes

In the same way as described for object identifiers we use regular definitions to express patterns of class identifiers. Regular definitions for class identifiers are also regular expressions

over the identifier alphabet. They are used in regular expressions for requirements tracing surrounded by square brackets ([,]). Any symbol of the form [defc], where defc is a regular expression over the identifier alphabet, is matched by any object whose class identifier matches defc.

Example 74 The regular definitions

```
    id  →  defines any symbol from the identifier alphabet
 clreq  →  id*Req
```

express that clreq is to be matched by any identifier ending in Req. Using this definition as a regular definition for class results that the symbol [id*Req] is matched by any object whose class identifier ends in Req. The regular expression

```
[id*Req] Derive R-001
```

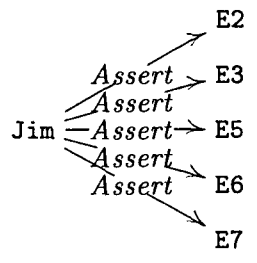
is matched, for example, by any object of class Req that is related to R-001 by Derive, or by any object of class TentativeReq that is related to R-001 by Derive. □

This form of regular definition may be used to trace (or to restrict the trace to) objects of a given class.

Example 75 In the configuration state of Figure 4.2, the regular expression

```
Jim Assert [Statement]
```

produces



as the resulting graph. □

4.6.3 Regular Definitions for Properties

Every class in TOOR is defined in the project specification being used to set up the project environment and any object created in TOOR should satisfy the specification of its class. For example, suppose that a project specification specifies a class Requirement as having an attribute priority taking values from a data type Priority (which may be high, medium,

`low`). Thus, using this project specification, any object of class `Requirement` created in TOOR should have the value of its `priority` attribute as `high`, `medium`, or `low`. Regular definitions for properties uses FOOPS axioms to specify properties that objects should satisfy in order to be considered to match the definition. They use FOOPS axioms to evaluate objects' attributes and methods. The general form of a regular definition for property is

`[[C x if` ⟨*FOOPSaxiom*⟩`]]`

where `C` is a class identifier, ⟨*FOOPSaxiom*⟩ is a FOOPS axiom, and `x` is a letter that may be used in ⟨*FOOPSaxiom*⟩ as a place holder for objects of class `C`. For each object being considered to match a regular definition for property, TOOR substitutes the object's identifier for `x` in ⟨*FOOPSaxiom*⟩ and submits the axiom for evaluation in FOOPS. If the result of the evaluation is the boolean value `true`, then the object is considered to match the regular definition.

Example 76 The regular expression

```
TR005-tp Derive [[TentativeReq x if priority(x) == high]]
```

is matched by an object identified by `TR005-tp` related by `Derive` to any object of class `TentativeReq` having a high priority, i.e., to `TentativeReq` objects for which the axiom `priority(x) == high` evaluates to `true`. TOOR evaluates the axiom for every object capable of matching the regular expression pattern. Searching the configuration state of Figure 4.2 TOOR evaluates `priority(TR051-tp) == high`, `priority(TR052-tp) == high`, and `priority(TR053-tp) == high` because `TR051-tp`, `TR052-tp`, and `TR053-tp` are all objects of class `TentativeReq` related by `Derive` to `TR005-tp`. If only `TR052-tp` has a high priority, then the graph

`TR005-tp` —*Derive*⟶ `TR052-tp`

is the result of tracing with this regular expression. □

The FOOPS axioms used in regular definitions for properties are evaluated by FOOPS against the definitions of the project specification. Therefore, they should respect these definitions. For example, using an attribute or method name not defined for the class `C` in a regular definition for property causes FOOPS not to parse the expression in ⟨*FOOPSaxiom*⟩. In these cases FOOPS issues an error message which is also sent to TOOR. If TOOR receives any value other than the boolean value `true` it considers the axiom to evaluate to `false`.

4.6.4 Expressing Composition of Relations

The use of regular definitions for class identifiers allows the expression of relational composition in a simple way. To specify that, for instance, requirement `reqA` is related to requirement `reqB` by a composition of relations `R` and `S` amounts to specify that `reqA` is related to some object `o` by `R` and that `o` is related to `reqB` by `S`. However, we want to express this using the more familiar notation `reqA R ; S reqB` where `R ; S` stands for the composition of `R` and `S`. This is achieved by means of the following parse transformation on regular expressions:

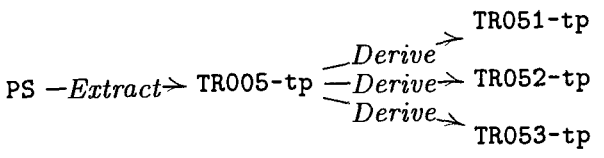
$$; \Rightarrow \texttt{[ToorObj]}$$

Whenever a semicolon appears in a regular expression TOOR replaces it by `[ToorObj]`, which is a regular definition for classes that is matched by any object since all objects are also of class `ToorObj`. For example `reqA R ; S reqB` is transformed to `reqA R [ToorObj] S reqB`.

Example 77 The regular expression

```
PS Extract ; Derive [Requirement]
```

is matched by an object identified by `PS` related to any object of class `Requirement` by a composition of relations `Extract` and `Derive`. The graph



is the result of using this regular expression to search the configuration state of Figure 4.2. □

Observe that it is possible to express the composition of several relations. For example, the parse transformations for `R ; S ; T` are

$$\texttt{R ; S ; T} \Rightarrow \texttt{R [ToorObj] S ; T} \Rightarrow \texttt{R [ToorObj] S [ToorObj] T}$$

4.6.5 Expressing Iteration of Relations and Objects

We may use regular definitions and the iteration operator to express composition of relations involving the repetition of the same relation symbol. For example, to find objects related to `PS` and to `SPEC-trace-back` by any sequence of relations we may use the following regular expression:

```
PS ToorRel ([ToorObj] ToorRel)* SPEC-trace-back
```

This can also be expressed by

```
PS ToorRel (; ToorRel)* SPEC-trace-back
```

but we want to be able to write the relation symbol only once, in a more intuitive way. This cannot be done with the star operator, like in `PS ToorRel* SPEC-trace-bacl`, because an expression like `R*` expands to ε (the empty string), `R`, `RR`, `RRR`, etc. None of these strings corresponds to a valid string of our grammar and will match no object. To introduce a parse transformation changing `R*` into a new expression where the symbol `ToorObj` would be placed between the relation symbols does not work. The empty string contained in `R*` would always make possible the side by side occurrence of two relation symbols and to strip off the empty string from the transformed expression could alter the original intention of `R*`. However, the use of a restricted form of iteration allows us to implement the necessary parse transformation. `R+` represents the positive closure of `R`, i.e., the set of strings consisting of *one* or more repetitions of `R`. We define the following parse transformations:

```
R+  ⇒  R ([ToorObj] R)*
O+  ⇒  O (ToorRel O)*
```

where `R` is a relation class identifier and `O` is an object class identifier. The above transformations are intuitively natural and significantly increase the expressiveness of our language.

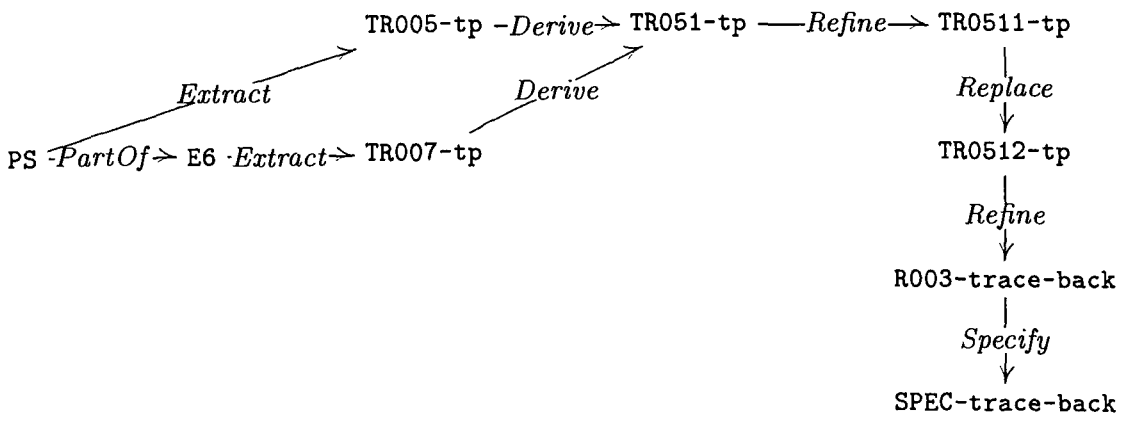
Example 78 To find the objects related to `PS` and to `SPEC-trace-back` by any sequence of relations is then expressed just by

```
PS ToorRel+ SPEC-trace-back
```

which is expanded to

```
PS ToorRel ([ToorObj] ToorRel)* SPEC-trace-back
```

and results in



□

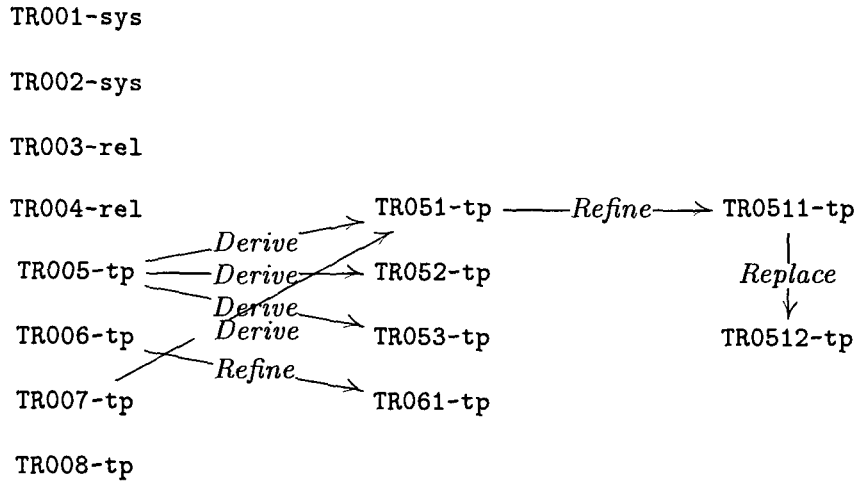
Example 79 To find all tentative requirements together with all relations or composition of relations between them is expressed by

```
[TentativeReq]+
```

This regular expression expands to

```
[TentativeReq] (ToorRel [TentativeReq])*
```

and results in



□

4.6.6 Expressing Exclusion of Objects and Relations

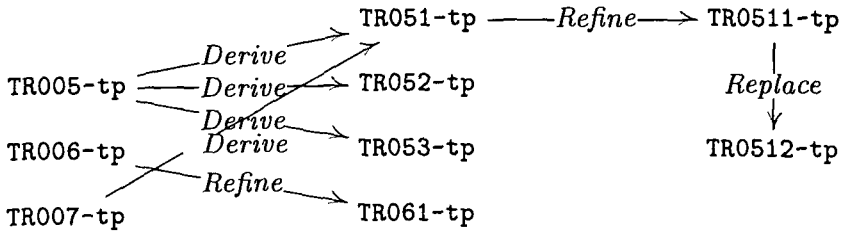
TOOR allows the use of negation (∼) and intersection (&) operators in regular expressions for requirements tracing. This makes possible the expression of things like "all objects related to req-A except those related by Derive" which would be written as

```
[ToorObj] ToorRel req-A & ∼([ToorObj] Derive req-A)
```

Example 80 In the previous example if we wanted to get only the related tentative requirements, it would suffice to exclude the unrelated ones by means of the following regular expression:

```
[TentativeReq]+ & ∼[TentativeReq]
```

The resulting graph would be



□

The use of an extended set of operators in regular expressions can make the construction of the underlying (regular expression) automaton very inefficient. For regular expressions containing only the regular operators of concatenation (.), union (|), and iteration (*,+) TOOR builds an ε-free NFA using the Glushkov construction [48, 19]. It is shown [19] that a Glushkov automaton can be constructed in a time quadratic in the size of the expression, and even in linear time for deterministic expressions. However, the Glushkov construction, as well as other constructions [11, 140] does not handle the additional operators of intersection and negation. To construct a NFA automaton recognizing the language denoted by a regular expression containing these operators (and indeed any boolean function based operator) we use the more general construction based on derivatives of regular expressions [20]. Derivatives of regular expressions do not apply directly to regular expressions in TOOR because of the use of regular definitions. Regular definitions in TOOR are regular expressions over a different alphabet: the identifier alphabet. Therefore, it is necessary to use a new set of rules for computing the derivatives of regular definitions with respect to symbols from the alphabet of object and relation class identifiers. These rules together with details of the implementation of derivatives in TOOR are presented in Appendix A. The user should exercise care when using the additional operators of intersection and negation since derivatives of regular expressions may be inefficient to compute, particularly if the alphabet is large.

4.6.6.1 Regular Definition for Negation

TOOR determines how to construct the regular expression automaton by looking in the expression for the additional operators & and $\sim$. In case they are not found the Glushkov construction is selected. Nevertheless, the user may use a restricted type of negation whilst maintaining the Glushkov construction.

The user can express that objects of a given class should be excluded from matching a pattern of objects and relations by means of a regular definition surrounded by exclamation marks (!). Any symbol of the form `!defneg!` where `defneg` is a regular expression over the identifier alphabet is matched by any object whose class *does not* matches `defneg`.

Example 81 One can express "all objects related to req-A except those related to req-A by Derive" using the following regular expression:

```
[ToorObj] !Derive!  req-A
```

Configurations of objects and relations of the form `d1 Extract req-A` and `d1 Refine req-A` will match this pattern since in both cases the relation instances are not of class `Derive`. □

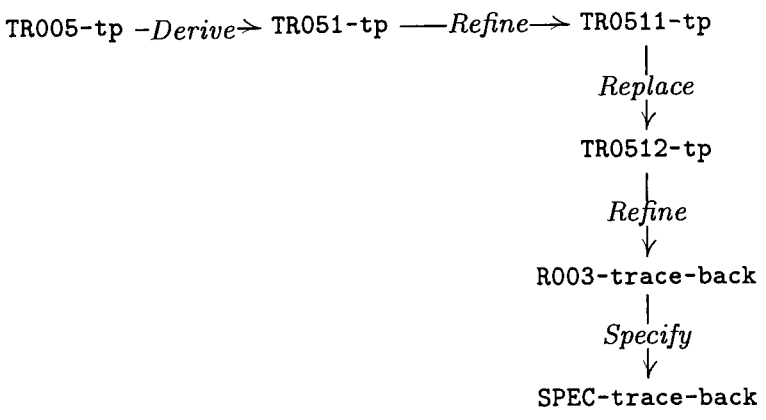
Since `!defneg!` is treated just as a symbol with a special matching procedure, the regular expression containing it is not regarded as being extended with additional operators.

4.7 Restricting the Tracing Space

Regular expressions are used to search a configuration state of objects and relations by finding strings in the associated configuration automaton matching the regular expression pattern. The result is also given as a configuration state: the configuration state extracted from the resulting intersection automaton. The resulting configuration state is always (except for searches like `[ToorObj] ToorRel+ [ToorObj]`) smaller than the original one. TOOR uses this fact to provide efficient tracing of objects by allowing regular expression searches to use the configuration state resulting from previous searches.

When submitting a regular expression to trace objects the user should give a base configuration state in which the trace will be performed. By default the base configuration state is the configuration state containing the objects and relations of the whole project but the user may select the configuration state that resulted from a previous search as the base configuration state. The objects and relations considered to match the pattern in the regular expression are only those in the base configuration state.

Example 82 After tracing with the regular expression `PS ToorRel+ SPEC-trace-back` and getting the configuration state of Example 78 as a result, the user may realise that requirement `TR005-tp` should be the starting point of a new search. Selecting the last configuration state as the base configuration and tracing `TR005-tp` forward with `TR005-tp ToorRel+ [ToorObj]` results in



Observe that if the base configuration for tracing was the project's configuration state (Figure 4.2), the result would be different because more objects would match the pattern. □

The user may browse and inspect the objects and relations in a configuration state resulting from a regular expression tracing. This may provide additional insights for further

searches and even suggest using a different trace mode, e.g., tracing according to the module structure starting from an object in the resulting configuration state. At any time the user can restore the base configuration to that of the project's configuration state.

4.7.1 Using Modules to Restrict the Tracing Space

The tracing space can also be restricted to a given module or set of modules. In this case the objects and relations that are considered to match a regular expression pattern are only those registered or being used in the selected modules. This is useful when one wants to restrict the tracing to certain parts of the project. For example, if an error is found, during the coding phase, in a certain source code module it may be advisable to trace it first looking for relations among other source code objects and to go back at most to specification objects. If this is the case and if the project is structured in a way that these objects are registered in specific modules, then the appropriate modules may be selected to restrict the tracing. Another example would be the tracing of requirements objects excluding those objects registered in modules created only to hold tentative requirements. The choice of which modules to use to restrict a trace depends on the purpose of tracing and, of course, on the way a project is structured.

Restricting the trace to selected modules has the advantage of giving additional freedom in the way regular expressions are written. One may use expressions like

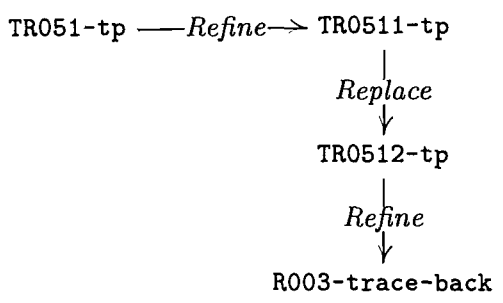
```
[ToorObj] ToorRel+ objid
```

and be sure that no object outside the selected modules will match the pattern.

Example 83 Suppose that the project described in Section 4.1.2 (whose configuration state is shown in Figure 4.2) has modules `SOURCE`, `REQUIREMENT`, and `SPECIFICATION`. The `SOURCE` module is used to register the `People`, `Statement`, and `Document` objects; the `REQUIREMENT` module is used to register the `TentativeReq` and `Requirement` objects; and the `FOOPSmod` and `FOOPStext` objects are registered in the `SPECIFICATION` module. Restricting the trace to module `REQUIREMENT` and tracing requirement `TR051-tp` forward with

```
TR051-tp ToorRel+ [ToorObj]
```

results in



The objects in SPECIFICATION which are also related to TR051-tp do not appear in the result.
□

4.8 Discussion

The use of regular expressions for tracing is a powerful instrument to narrow large spaces of objects and relations in a focused way. By giving the overall pattern of objects to be matched the user can make sense of what is important to retrieve, and leave out the objects that are not relevant to a particular search. The mechanism to match objects is not restricted to the use of classes and object identifiers. Regular definitions greatly increases the expressiveness of regular expressions and make possible the specification of patterns in a flexible way. Because regular definitions are regular expressions themselves their use is equivalent to the use of patterns inside patterns. The user can specify that the match of an object to a given symbol is to occur according to a special pattern rather than according to the object identifier or the object class identifier. This can be used in connection with a naming convention for object identifiers to retrieve objects with similar names. It can also be used to specify symbols that are to be matched by objects of different classes or by objects that do not belong to a given class. Also, objects can be specified according to their contents by the use of FOOPS axioms to evaluate object's attributes and methods.

The different ways in which a large space of objects and relations can be searched and the fact that the pattern matching procedure is not fixed (it varies according to the regular definitions used) makes tracing with regular expressions adaptable to different situations. Also, the ability to store regular expressions promotes their use by people having different skills and supports administrative or monitoring tasks. For example, the tracing of requirements that are late (in being allocated to some part of the design) or the tracing of requirements having a certain priority value can be stored for regular use. Tracing with regular expressions can be constrained to smaller configurations resulting from previous search or to particular modules. This is a flexible way of performing incremental tracing and it helps reasoning about tracing results. The integration of regular expression tracing with other forms of tracing such as browsing and interactive tracing through module structure makes TOOR an extremely versatile tool. The user can select the more appropriate form of tracing depending on context and can switch from one form to another as convenient.

Chapter 5

Enhancing Traceability

The path from a problem statement to a Requirements Specification Document (RSD) and beyond is most of the time not a linear one: there is much going back and forth, e.g., involving hypotheses, false starts, experiments, dead ends, multiple paths, etc. Tracing in such a dynamic environment is seldom straightforward.

Deciding to trace something involves some motivation. In a system with hundreds (and often much more!) of interrelated objects, getting all the information available may not be useful. There may be so much information that the configurations before and after tracing are almost the same. Also, the purpose of tracing may be viewed as getting related information to fulfill the motivation for tracing. In this respect the form in which information can be retrieved and traced becomes an important point to consider. Restricting tracing to certain modules and selective tracing through the use of regular expressions are necessary in cases where one wants to reduce the space of tracing. However, the criteria to be used in selecting information may be not so clear. A user can often get valuable insight by 'browsing around', guided by experience and background.

Traces in TOOR are modelled by relations among objects. The traceability model of TOOR provides a convenient and flexible way to specify, store, and retrieve traces as relations. However, the existence of mechanisms to precisely define what information to trace and the existence of a structure to guide the tracing are not always enough to fulfill the motivation for tracing since it presupposes that one knows in advance what has to be defined and how to follow the information being traced. In some situations it is useful to have ways of browsing objects looking for information whose relation to the object being traced is not evident from the outset. The browsing mode of TOOR provides functionality analogous to browsing through a book: one can search for something seen, for example, somewhere near the middle of the book, probably at an even-numbered page.

Some limited browsing functionality is given by the interactive tracing through the module

structure of a project: the user can choose what is the next object or what is the next module to be considered in following the tracing path. However, TOOR has other, more flexible, ways to browse objects.

The arguments given in favor of browsing objects may be used to argue in favor of the availability of data in their original format. If one is looking for information that is not precisely formulated and if it is not certain how it relates to the object being traced, then valuable insight may be gained from inspecting data in their original manifestation. This points to the utility of hyper-media features associated to the tracing procedure. It is not the only reason; the need to re-interpret data is usually behind the need to trace some object and the fact that information is usually encoded in different formats also suggest the need for some hyper-media support when tracing objects.

This chapter presents the browsing modes available in TOOR and discusses its hyper-media features, through which the user can access data in their original format.

5.1 Browsing Objects

Objects may be browsed in TOOR in several different ways. First, every window presenting tracing results is a browsing window in the sense that the user may inspect the contents of any object in it. Therefore, while tracing an object through TOOR's module structure the user can click on any object shown in the window containing the trace results and inspect its contents. The same applies to the window showing the configuration state resulting from a regular expression trace; any of the objects and relations in it may be inspected by the user by just clicking on it. Alternatively, TOOR offers dedicated browse windows that may be used concurrently with any other window being shown.

The first of these browse windows is the *configuration state window* which presents the configuration of objects and relations in a graphical way. It is similar to the window resulting from tracing with regular expressions. In particular, the objects and relations are shown in the same way: as a graph with objects as nodes and arrows indicating the source and target object for each relation instance. The difference is that the user can view the configuration state at any time and not only as a result of a regular expression search. The whole configuration state may be presented in this window but there are some options making the viewing of objects and relations more manageable in cases of a large number of objects. The user can choose to view only objects of a given class or of a given group of classes. The same applies for relation classes. This makes visual inspection more efficient. The user can select only `People`, `Interview`, and `Requirement` classes to be shown, or only relations of class `Derive`. After inspecting the objects shown in this restricted way new classes can be added and some may

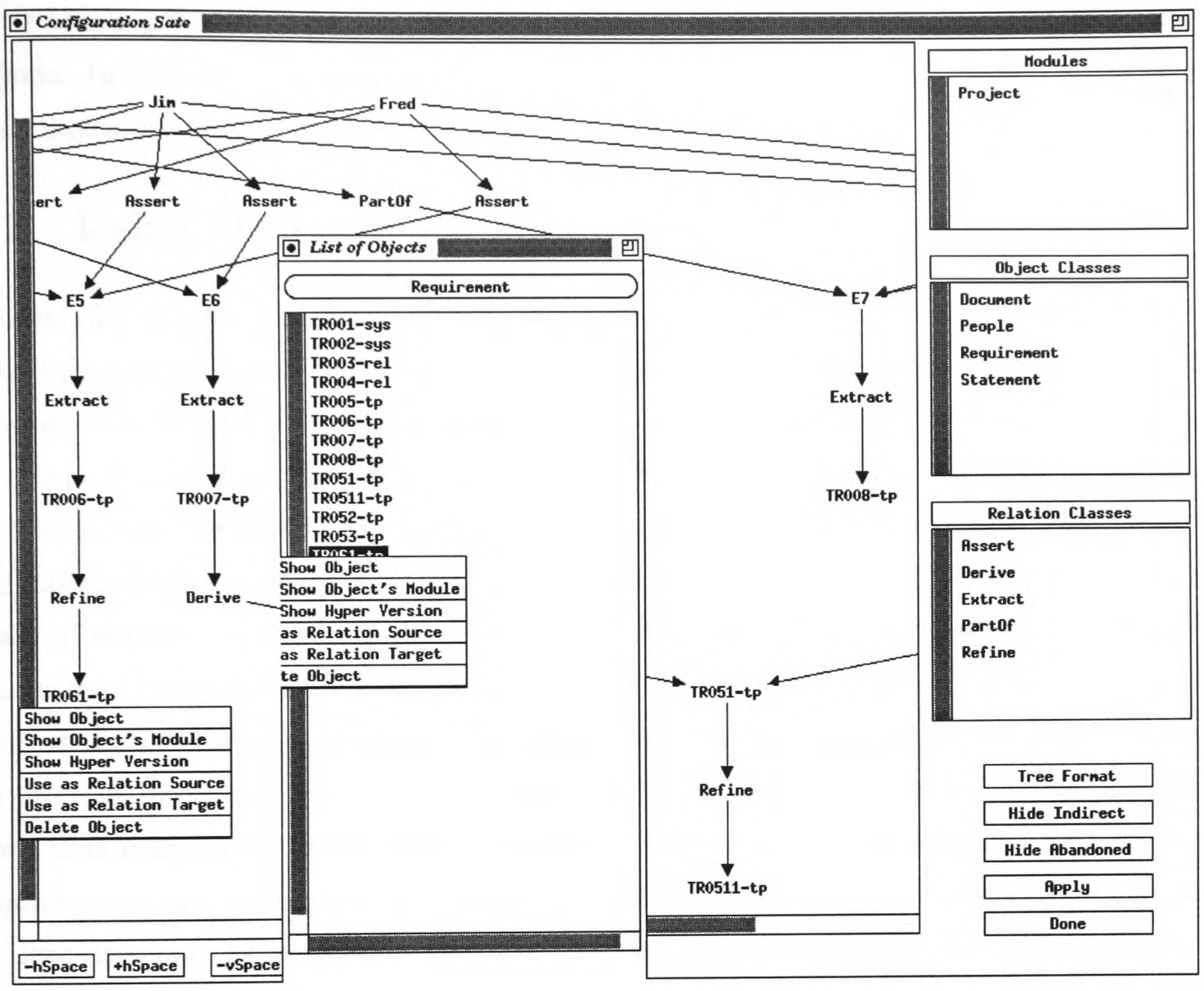


Figure 5.1: Browsing Windows in TOOR

be removed. For example, `People` and `Interview` may be removed and `Specification` may be added to give a forward view of the requirements. The user may have several configuration state windows active. This make possible the existence of different views at any time.

The second browse window is the *object list window.* The objects are shown in a list containing their identifiers. In the same way the user may choose to show only objects of a certain class or of a particular group of classes. In this case the objects may also be sorted in different ways, they may appear sorted by date of creation, date of modification, object identifier, class identifier, or class and object identifier. Again, the user may have several object list windows at any time.

Figure 5.1 shows a configuration state window and an object list window for the extended example of Chapter 4. The object list window contains a list of the registered requirements. The menu under requirement `TR061-tp` in both windows contains options that allow the user to inspect either the object or the object's module.

Browsing objects may be further restricted to objects of a given module or to objects of

a particular group of modules. The user may also browse the TOOR modules instead of the objects. In this way the module structure is presented in a graphical form and the user may open any module to view the objects in it.

5.2 Using Hypermedia Features

Traceability is greatly enhanced if one can access data in their original format. The interpretation process leading from raw data to requirements can be re-assessed effectively only if the original data is available. Otherwise, just having the interpreted results makes it impossible to compare and check the decisions made and to validate the assumptions against the data on which they are based. The re-interpretation of previous results and the understanding of the interpretation process leading to a given set of requirements or design artifacts are at the heart of many traceability queries. Why is a given requirement expressed in a certain way, where does a requirement come from, what is the basis for a given requirement, in which context is a requirement placed, what changes can a requirement undertake and still preserve its original intention, how may a change affect other related objects – these are all questions that may be answered with increased confidence if one has access to data from which requirements are elaborated. Of course, this also applies to objects other than requirements.

Requirements have different sources. The elicitation process may involve interviews, analysis of existing documents or systems, analysis of work practices through observation and video analysis techniques, simulation of environment models, prototyping of systems features, etc. The raw data from which requirements are elicited are an important part of the elicitation process and most requirements (traceability) systems include some way of referencing them. In TOOR the raw data may be accessed in their original format. Therefore, the user can playback a video recording used as part of a video based analysis, hear a discussion from a recorded interview, see the animation associated with some concept, and, generally, retrieve the contents of a file using a viewer appropriate to the file's format.

TOOR allows the use of files as attribute values. There is a standard `T$File` datatype providing basic file manipulation. When filling in the template for a particular object whose class has a `T$File` valued attribute, the user should give the file name corresponding to that attribute. The content of the file (the attribute value) is viewed through a viewer appropriate to the type of the file given as the attribute value. In every template showing an object there is a *hyper* button that, when clicked on, will show the hyper version of the object containing a link to retrieve the file associated to any `T$File` valued attribute. At the moment MOSAIC is used as the hypermedia browser. Therefore, the types of files that maybe viewed in TOOR are those for which there is an appropriate viewer from MOSAIC. However, TOOR can easily

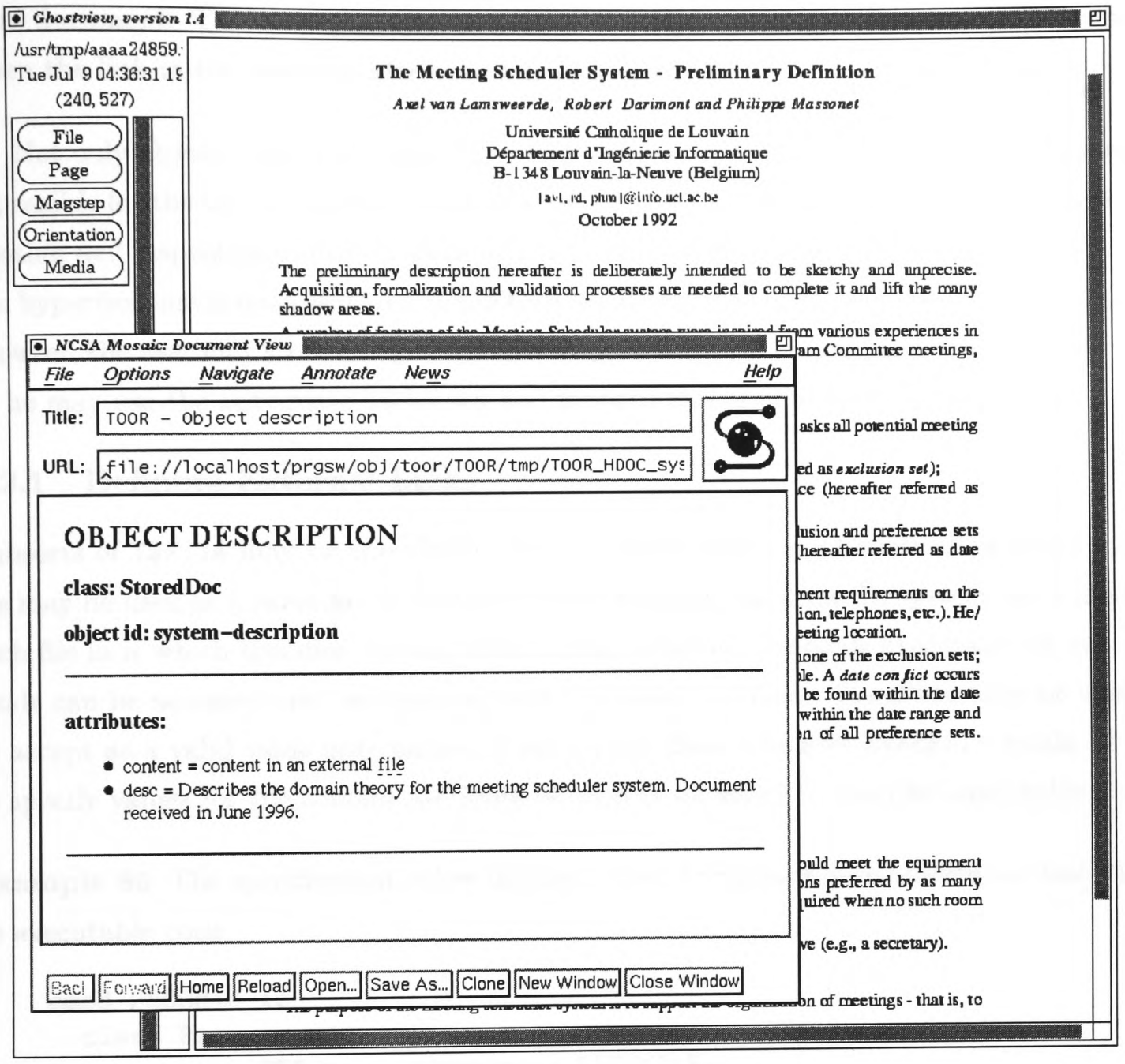


Figure 5.2: Hyper Version of TOOR Objects

accommodate other browsers since the hypermedia version of its modules and objects are kept in the HTML format accepted by most existing browsers.

Example 84 The specification below specifies stored documents as objects whose content attribute is given by the file containing the document itself.

```
omod DOCUMENT is
  class StoredDoc .
  protecting TEXT .
  protecting FILE .
  at desc    : Document -> Text .
  at content : Document -> T$File .
endo
```

Figure 5.2 shows the hyper version of a stored document `system-description` as a MO-

SAIC document. The viewer GHOSTVIEW for PostScript documents was activated by *mosaic* when the link in the `content` attribute was clicked on. □

Not only objects have a corresponding hyper version but also TOOR modules. This makes it possible for the user to browse the objects in a hypertext like way, going from one module to another and inspecting objects in each one. This may be interesting because it makes available the hypertext navigation features of the browser being used. For example, depending on the browser, the user may store a page corresponding to a certain module and reload it afterwards or he may use the automatic backward and forward chains of already visited pages.

5.2.1 Defining External Links

Subsorts of `T$File` may be specified to give an additional way of controlling which kind of file may be used as a value for an attribute. For example, the UNIX file system has a mode for each file in it which specifies, among other things, whether the file is executable or not. This mode can be accessed via a system call and, therefore, a subsort `ExecFile` may be specified to accept as a valid value only names of executable files. The sort `ExecFile` would be used to specify values for the content attribute of objects intended to describe executable code.

Example 85 The specification below defines a class `Program` whose objects are intended to be executable code

```
omod PROGRAM is
  class Program .
  protecting TEXT .   protecting EXECFILE .
  at prg_desc : Program -> Text .
  at prg_code : Program -> ExecFile .
endo
```

The module `EXECFILE` contains the definition of the `ExecFile` sort as a subsort of `T$File`. The sort `T$File` is a built-in sort which makes use of LISP functions to decide if a given value is valid or not. The sort `ExecFile` should also be a built-in sort whose LISP functions determine, given a file name, whether the named file has an executable mode or not. The definition of subsorts of built-in sorts is explained in the FOOPS manual [123]. □

Any sort of file may be specified in this way provided it has some distinctive characteristic that may be tested by a LISP function. All such sorts should be defined as built-in subsorts of `T$File`. When creating the hyper-media version of objects as HTML documents, TOOR makes links between attribute's contents and external files only for those attributes whose values are of sort (or subsort of) `T$File`.

We should stress the point that this type of definitions, and generally the definition of a project specification, is intended to be made by a system administrator proficient in the use of FOOPS. Once a project specification is defined, the use of TOOR to register objects is very simple and intuitive. Browsing objects and tracing them interactively or using simple form of regular expressions is also easy. More sophisticate use of regular expression, involving regular definitions and FOOPS axioms, requires some training.

5.2.2 Tracing Contents of Attributes

The hypermedia features of TOOR increases the effectiveness of requirements traceability by making it easier to reinterpret data on which objects are based. A regular expression like

```
[Document] ToorRel reqA
```

would produce the documents that are related to requirement `reqA`. If any of the documents in the resulting configuration state have a file valued attribute, the user could click on it and access the file contents. This could be the recording of an interview or a diagram providing contextual details for the document description. However, although valuable, access to the file contents of a given object may occur only after the object has been traced in some way, i.e., the object has to be accessed first for the user to click on it. In many cases one may want to be more selective when tracing and get directly those objects whose file contents contain some pattern. For example, when analyzing a requirement such as

> **reqA** *The transmission speed of data packets of type A should be no less than 10 packets/second.*

the user may want to trace it back to any document whose text contains the expression "data packet" or "packet type A". In TOOR this may be done through the use of FOOPS axioms in regular expressions. The tracing of the requirement above in the way just described would be expressed as

```
[[Document x if srchdoc(x,''data packet'')]] ToorRel reqA
```

The `srchdoc` in the regular definition for property is a method specified for the class `Document` which returns `true` if its second argument is contained in the file associated to the content attribute of a document `x`. The next example shows how this may be done.

Example 86 In the standard project specification there is a sort `FileTxt` intended to be the sort for text files. It is specified as a subsort of `T$File` in the following module:

```
fmod TXTFILE is
  sort FileTxt (t$filetxt_is_sort_token, t$filetxt_create_sort,
                t$filetxt_print_sort, t$filetxt_is_sort) .
  protecting FILE .
  subsort FileTxt < T$File .
  fn srchtxt : FileTxt Id -> Bool .
  var S : Id .
  var F : FileTxt .
  beq srchtxt(F,S) = (srchtext F S) .
endf
```

The functions t$filetxt_is_sort_token, t$filetxt_create_sort, t$filetxt_print_sort, and t$filetxt_is_sort in the definition of FileTxt are LISP functions required for built-in sorts. They provide the necessary code for FOOPS to manipulate values of the built-in sort. The sort T$File is defined in the module FILE and the operator srchtxt is specified by a built-in equation (beq) which executes the lisp function srchtext. The lisp function srchtext receives a file name F of sort FileTxt and an identifier S, opens the file and returns true if the identifier is contained in the file.

The following specification for documents use the sort defined above to specify the values of the attribute doc_content.

```
omod DOCUMENT is
  class Document .
  protecting TXTFILE .
  protecting TEXT .
  at doc_name    : Document -> Text .
  at doc_desc    : Document -> Text .
  at doc_content : Document -> FileTxt .
  me srchdoc     : Document Id -> Bool .
  var S : Id .
  var D : Document .
  ax srchdoc(D,S) = srchtxt(doc_content(D),S) .
endo
```

The method srchdoc when applied to an object D with an identifier S as argument call the operation srchtxt on the file name given by the doc_content attribute. □

This kind of tracing, searching for patterns stored in external files, may be used to follow 'links' not explicitly registered in TOOR. A document containing a keyword or expression need not be explicitly related to a requirement or any other object to be traced. The regular expression

```
[[Document x if srchdoc(x,``data packet'')]]
```

retrieves all documents containing the expression "data packet" whether they are related to the requirement originating the need for the search or not. This may be regarded as a kind of tracing implicit links between the intention or purpose of trace and the objects containing information that matches the pattern in which the intention is expressed. The definition of searching methods associated to the specification of the classes used in a project may be applied to any kind of attribute; not only to `T$File` valued attributes. They also may be as complex as desired. For example, with appropriate definition of sorts and search operators in the project specification one can trace objects based on patterns given in several different formats. For example an operator `srchsound`, given a file containing a digitised sound, could search a document for occurrences of similar sounds.

5.3 Discussion

In the initial phases of software development information is gathered and used in a variety of different formats. There may be, for example, lots of verbal information resulting from interviews and other forms of eliciting requirements such as protocol analysis; there may be plenty of pictorial information resulting from the use of graphs and diagrams to express ideas; and there may even be visual information captured through the use of video analysis techniques. For a detailed account of various techniques of eliciting requirements and the type of information that may be gathered in using each technique see [60].

Information is not only expressed in different ways, it is also encoded in different formats. For example, a text may be encoded as an ASCII, a DVI, or a PostScript file, each one of these formats requiring an appropriate 'viewer' to access the information it encodes. This is true for all phases of system development. An example, from later phases would be that of binary and source codes of a program.

In this chapter we have argued that the ability to refer to data in their original format greatly enhances traceability by providing the means for a more accurate reassessment of whatever was taken into consideration when relating one object to another. In this chapter we also presented the browsing mode of TOOR as an additional functionality to help tracing in situations where little information of what has to be traced is possessed.

Chapter 6

Tracing Requirements Evolution

Getting requirements right is seldom straightforward: it may be necessary to negotiate conflicts among the interested parties, to clarify the meaning of a given term used in different contexts, and to carefully analyse tradeoffs. Moreover, new relations among requirements, people, documents, concepts, and other objects may appear in the process. Change is inevitable and unending in large, complex systems. Changes to requirements arise not only from changes in the social context of the system, but also from improved understanding of constraints and tradeoffs as system development proceeds. Moreover, the system and its requirements *co-evolve* with the social context in that each affects the evolution of the other.

Requirements evolve throughout the entire lifetime of a project. A Requirements Specification Document (RSD) states a set of requirements agreed at some specific time; it may represent a landmark in the development effort, or a contract between the interested parties. In any case, producing an RSD does not freeze requirements; as development proceeds, new ideas, design diagrams, specifications, code, etc., will be produced, generating new objects and relations, and inevitably modifying the requirements.

All this happens in a non-linear dynamic way. However, the evolution of objects from requirements sources to requirements to design to code, and generally to any object taking part in the process is dealt with in a uniform way in TOOR: classes are declared for each kind of object we wish to control, and relations are defined between them. For example, requirements change, abandonment, and deletion are defined as relations on requirements. Also, the path from requirements to specification to design to code and so on is modelled by relations on the appropriate trace units. Therefore, changes occurring in the context of software development are managed by the appropriate definition of their type in terms of objects and relations. For those changes that were not specified previously, the user can modify the specification for the project and incorporate them at anytime. The project specification can also be modified to test hypotheses about relations or to change the way objects are related.

In this chapter we present the facilities available in TOOR to trace requirements evolution. First we describe the replacement and abandonment of requirements by means of an example of actual software development. Then we give details of the relations used in TOOR to keep track of these kinds of modifications. This is followed by a discussion of how to trace object modifications.

6.1 An Introductory Example

The Meeting Scheduler example in this section is taken from [143]. It is chosen because the authors report on the 'actual' elaboration of its specification. Particularly, they present many situations where it was necessary to backtrack to previous points, to undertake parallel development, and to modify previous definitions. Here we concentrate on the dynamics of the development, as reported, showing the creation of objects and relations and their re-assessments. For the presentation of this example we sometimes maintain the formal notation used by the authors to formalize the concepts involved. This allows us to follow the same justifications for the steps taken in the development of the system.

The KAOS method was used to develop the specification for the Meeting Scheduler System. The objective of the KAOS method is to specify systems in terms of the objects concerning the system domain and in terms of the agents and the actions they should perform in order to satisfy the constraints determined from the system's goals. In short, goals may be reduced to other goals and are operationalized into constraints which are satisfied by actions and objects. Objects concern goals and agents, which are a sort of object, perform actions. The details of the method are not given, but the transformations are explained so that we can have a good idea of how it works.

6.1.1 Registering Objects and Relations

Specifications developed using the KAOS method can be effectively modeled in TOOR by introducing classes for each of its concepts. In this example we only discuss the configuration state of objects and relations resulting of using TOOR to model KAOS concepts. A fuller discussion of using TOOR to support traceability in other methods is given in Chapter 7.

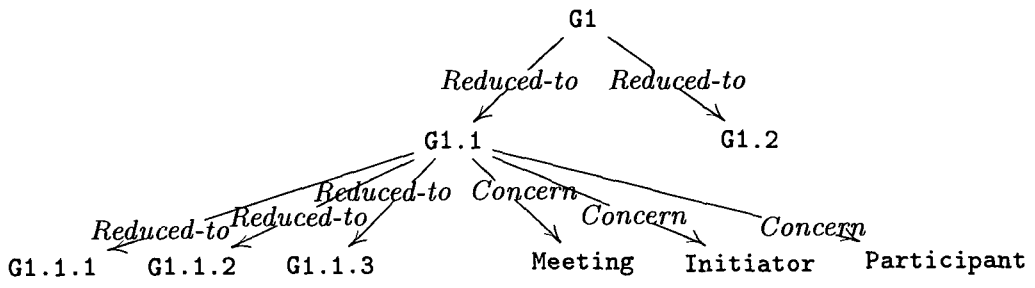
Objects of class `Goal` are defined in the project specification to have an attribute `infdef` for its informal description and an attribute `fmldef` for its formal definition. The first goal object is described informally as

G1(MeetingScheduler)
infdef: *The purpose of the meeting scheduler system is to support the organization of meetings – that is, to determine, for each meeting request, a meeting date and location so that most of the intended participants will effectively participate.*

Goal G1 is reduced to goals G1.1 and G1.2, and goal G1.1 is further reduced to goals G1.1.1, G1.1.2, and G1.1.3. The informal definition for these goals are omitted since their names give enough information of what they are about.

G1.1(Meeting – Request – Satisfied)
G1.2(Maximize – Number – Participants)
G1.1.1(Participants – Constraints – Known)
G1.1.2(Meeting – Planned)
G1.1.3(Participants – Notified)

The reduction process is registered in TOOR as a relation Reduced-to between goals. Also, the relevant objects for each goal are identified and registered in TOOR, as well as the relation Concern between goals and objects. The configuration state at this moment is given by



6.1.2 Modifying Already Registered Objects by Edition

The process of analysing goals for reduction and to find out what are the likely objects of concern frequently leads to a more precise understanding of the goals. As a result, definitions of objects may be modified. The goal G1.1 has its informal definition modified in this way and also a formal definition is given for it. The attributes infdef and fmldef are modified by editing their old values of which nothing is kept in the TOOR's database. Apart from the contents of objects, the configuration state does not change as a result of modifying G1.1.

G1.1(Meeting − Request − Satisfied)

infdef: *Every meeting request should be satisfied within some deadline associated with the request. Satisfying a request means proposing some 'best' meeting date/location to the intended participants that fit their constraints, or notifying them that no solution can be found with those constraints.*

fmldef:
$$(\forall r : \text{Initiator}, m : \text{Meeting})$$
$$\text{Requesting}(r, m) \wedge \text{Feasible}(m) \Rightarrow \Diamond \text{Scheduled}(r, m)$$

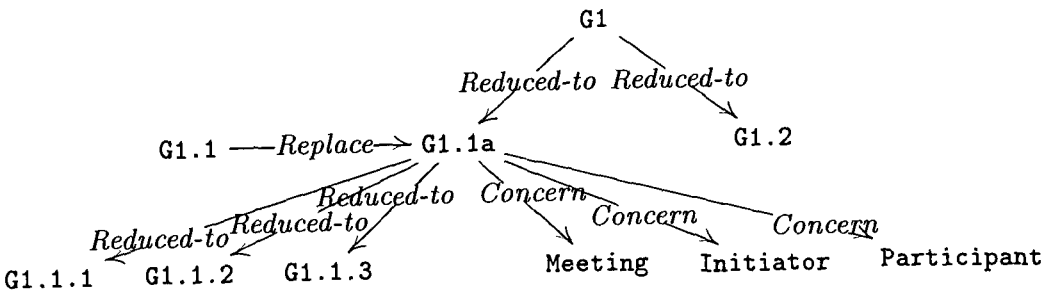
The formal definition above is a formulation in temporal logic of the informal definition. We use the same notation to ensure consistency between this example and the description of the development given in [143].

6.1.3 Modifying Already Registered Objects by Replacement

In KAOS the first formal description of a goal or object is usually taken from a knowledge base. From a goal's pattern, category, and type possible descriptions for it are retrieved and the best one is chosen (possibly) after some adaptation. In this example the process of finding an appropriate formal description for a goal is not registered in TOOR. The user only types in the final form chosen for the `fmldef` attribute. Formal descriptions of goals change as the analysis process continues. The goal `G1.1` has its formal description changed to include constraints about scheduling meetings and informing participants:

$$\begin{aligned}(\forall r : \text{Initiator}, m : \text{Meeting}, p : \text{Participant})\\ \text{Requesting}(r, m) \wedge \text{Feasible}(m) \Rightarrow \Diamond_{\leq R(r,m)d} \text{Scheduled}(r, m)\\ \wedge \quad \text{Invited}(p, m) \Rightarrow \Diamond_{\leq P(p,m)d} \text{Knows}(p, m)\end{aligned}$$

In this case, instead of just substituting the new formalization for the old one, it is decided to keep the old description for reference. This is made by creating a new goal `G1.1a` to replace the old goal `G1.1` which is being modified. Both goals remain in the TOOR database although the one being replaced is hidden from the normal use of TOOR. The relations involving `G1.1` are transferred to `G1.1a`. The new configuration state is given by



6.1.4 Evolving Object's Descriptions

In parallel the definitions of other objects are assessed and modified in a similar way: the new definitions substitute old ones if no trace of the old definitions are desired or are not felt necessary, or else new objects containing the new definitions are created to replace objects with old definitions by a `Replace` relation.

As part of the analysis process the formal definitions for goals `G1.1.1`, `G1.1.2`, and `G1.1.3` are used to prove the goal `G1.1a` follows from them. During the proof a debugging process also occurs and, as a result, it was found the a sub-goal was missing. The new sub-goal `G1.1.4` (Scheduler Available) reduced from `G1.1a` is incorporated in TOOR's database. Also, the the formal definition of `G1.1.1` given by

$$\begin{aligned}(\forall m : \text{Meeting}, p : \text{Participant}, s : \text{Scheduler})& \\ \text{Invited}(p,m) \ \wedge \text{Scheduling}(s,m)& \\ \Rightarrow \diamond_{\leq C(p,m)d}\text{Knows}(s, p.Constraints)&\end{aligned}$$

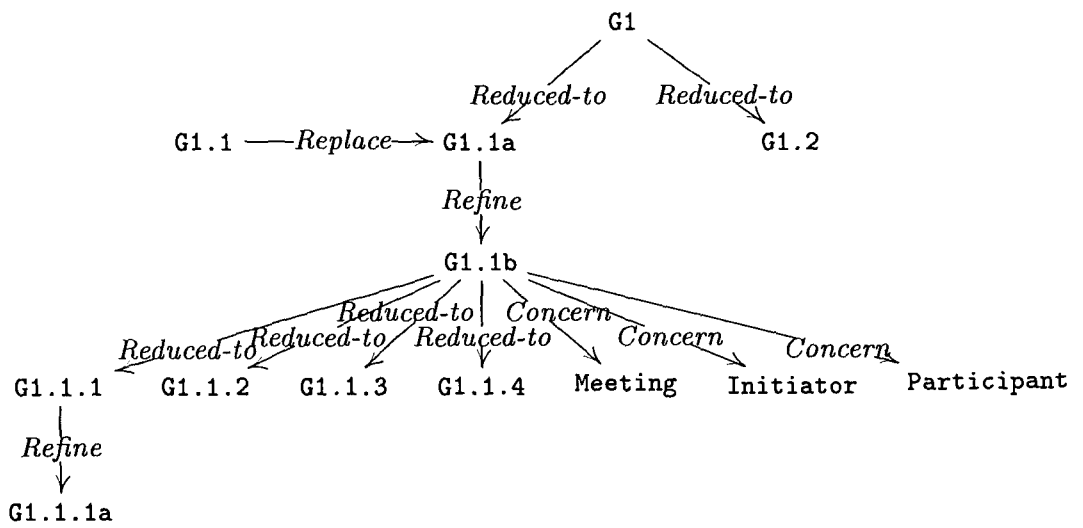
is found infeasible because one cannot suppose that participants will always inform their time constraints to the scheduler. Therefore, a new weaker definition is taken for `G1.1.1`.

$$\begin{aligned}(\forall m : \text{Meeting}, p : \text{Participant}, s : \text{Scheduler})& \\ \text{Invited}(p,m) \ \wedge \text{Scheduling}(s,m)& \\ \Rightarrow \diamond_{\leq C(p,m)d}\text{Knows}[(s, p.Constraints) \vee \neg\text{Expected(p,m)}]&\end{aligned}$$

This time a new goal object `G1.1.1a` (Participants-Constraints-Known) is created not to replace `G1.1.1` but to register the modification as a refinement documented in TOOR's database. The new goal `G1.1.1a` is registered in TOOR and related to `G1.1.1` by an instance of the `Refine` relation. The reassessment of goals may highlight the infeasibility of other previously defined goals. As an example, the goal `G1.1a` is also modified to be

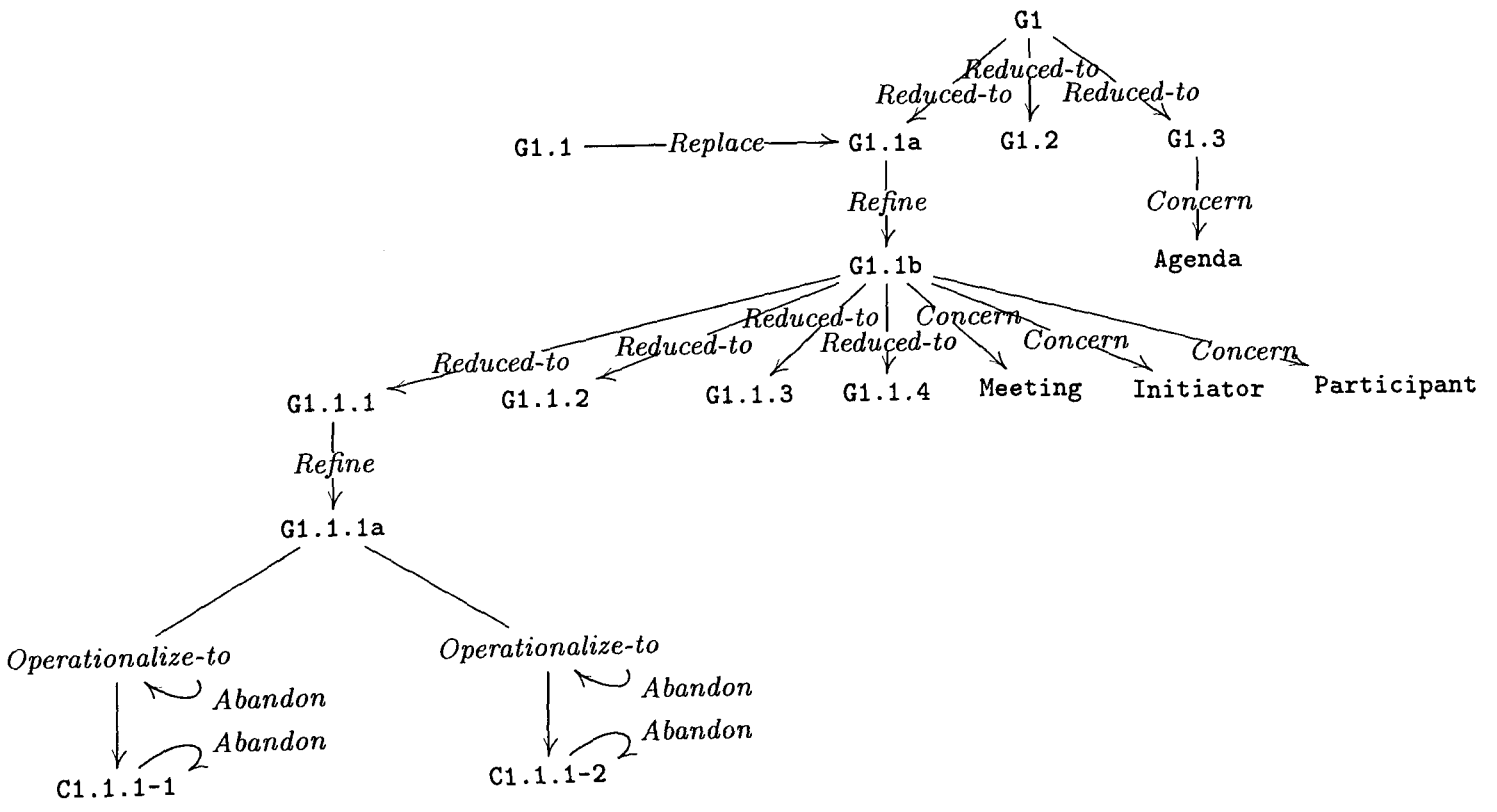
$$\begin{aligned}(\forall r : \text{Initiator}, m : \text{Meeting}, p : \text{Participant})& \\ \text{Requesting}(r,m) \Rightarrow \ \diamond_{\leq R(r,m)d}[\text{Scheduled(m)} \vee \neg\text{Feasible}(m)]& \\ \wedge\text{Invited}(p,m) \Rightarrow \ \diamond_{\leq P(p,m)d}\text{Knows}(p,m)&\end{aligned}$$

The new definition for `G1.1a` is also registered as a refinement by means of a new object `G1.1b` related to it by a `Refine` relation. The configuration state becomes



6.1.5 Trying Alternative Developments and Abandoning Objects

The constraints `C1.1.1-1` (Achieve-Participants-Constraints-Requested) and `C1.1.1-2` (Achieve-Participants-Constraints-Provided) are used to operationalize goal `G1.1.1a`. They are assigned to agents and should be ensured by objects and actions. In this case `C1.1.1-1` may be assigned to Scheduler and `C1.1.1-2` to Participant. However, this operationalization is just one alternative. A second option, also tried, consists of developing a goal `G1.3` about the interaction between participants and scheduler. The new goal concerns an object **Agenda** which is made available to the scheduler. Both ways are registered but eventually the option involving the agenda is chosen and the objects related to the first option are abandoned. The objects are abandoned by relating them to themselves by an instance of the **Abandon** relation. The configuration state after the abandonment of `C1.1.1-1` and `C1.1.1-2` is given by



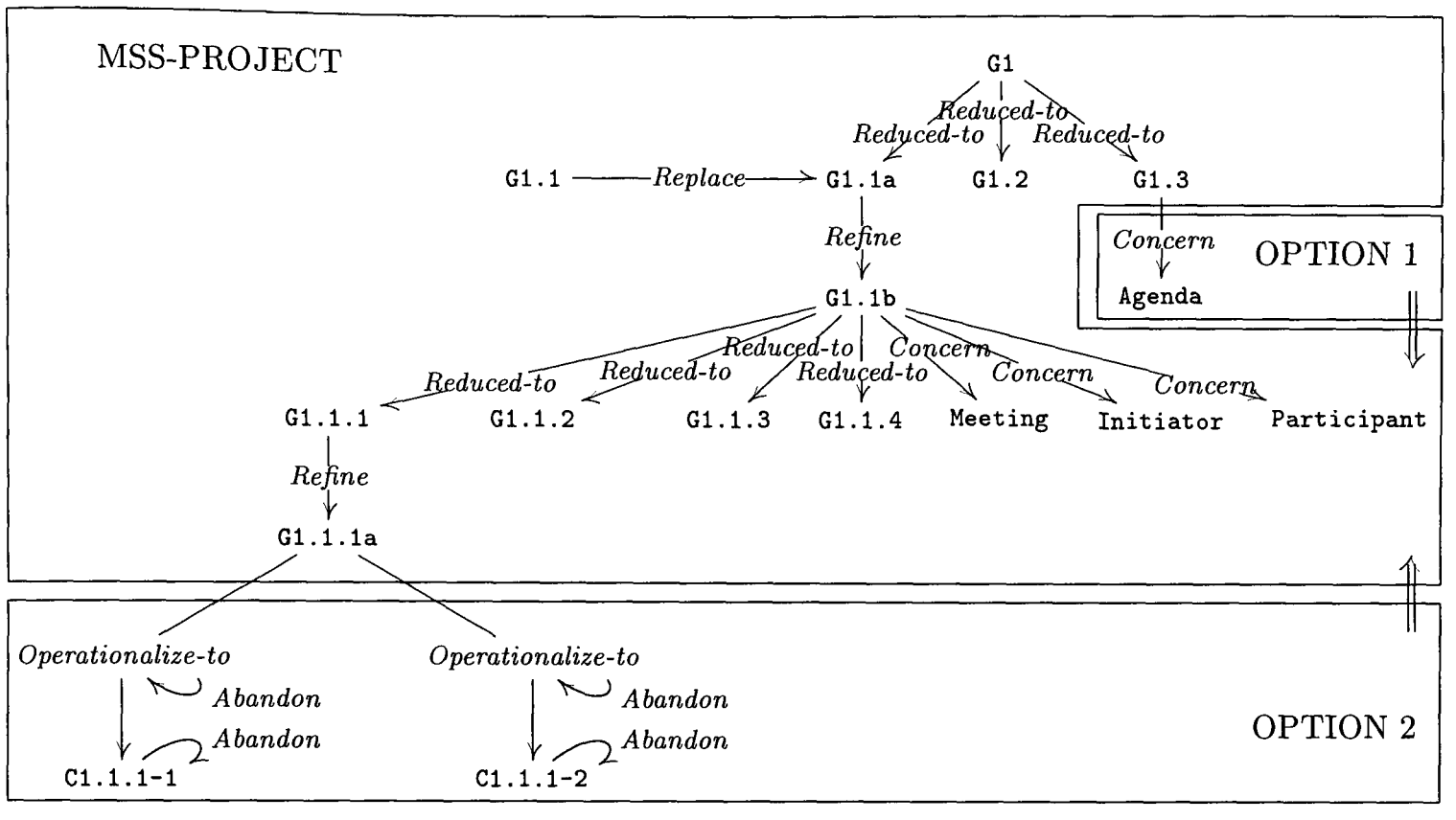


Figure 6.1: Using Modules to Delimit Alternative Developments

By abandoning the objects `C1.1.1-1` and `C1.1.1-2` all relations involving them are also abandoned. However, the objects remain in TOOR's database. They may be traced and retrieved if necessary. An option would be simply to delete the objects, thereby losing all trace of them, but the user could also make use of the structuring facilities of TOOR to develop both alternatives in separate modules. Figure 6.1 illustrates this option. A module `MSS-PROJECT` contains the objects of the basic development and modules `OPTION1` and `OPTION2` are created to hold objects associated with the alternative developments. Both modules, `OPTION1` and `OPTION2`, import `MSS-PROJECT` and may use any object registered in there, but one cannot access the objects registered in the other. Also, tracing may be restricted to the context of one module, `OPTION1` or `OPTION2`, so that objects registered in the other one are not retrieved.

6.2 Keeping Track of Evolution in TOOR

The modifications shown in the previous example were based on the actual steps reported in [143]. In particular, the change of informal and formal descriptions and the reassessment of previous points are in accordance with their description. The creation of new objects and the use of the `Replace`, `Refine`, and `Abandon` relations are additions made to illustrate the use of TOOR to keep track of project evolution.

Of course, in developing the system and performing the changes much tracing was done. Every change should be propagated downwards and upwards along the `Reduced-to`,

`Operationalize-to`, `Concern`, and other relations. Also, tracing backward and forward to retrieve related objects is necessary for a correct assessment of a possible modification. The actual tracing procedure depends on the situation: it may be through browsing and visual inspection, it may be through the modular structure of the project, it may be through regular expressions, or a combination of those modes. Rather than the tracing possibilities, what the example highlights is the way that project evolution may be registered in TOOR and made available for tracing. The following sections give details of the use of relations and objects' modification dates to keep track of modification, replacement, refinement, and abandonment.

6.2.1 Deriving and Refining Requirements

The process of deriving one requirement from another may be described by a relation called *Derive* and the refinement of requirements may be described by a relation called *Refine*. The original requirement is the source object and the derived (or refined) requirement is the target object of each instance of the *Derive* (or *Refine*) relation.

The difference between requirements derivation and requirements refinement relates to the notion of satisfaction for requirements. If a requirement is satisfied, then its derived requirements should also be satisfied; however, satisfying a derived requirement does not mean that the requirement from which it was derived is also satisfied. For refinement, the converse holds. If one requirement refines another, satisfying the refined requirement implies satisfying the original requirement. This notion of derived requirements differs from the one stated by Davis [33] and is closer to the concept of requirements partial dependence of Dasgupta [139]. However, as user-defined relations other meanings can be associated to them. The correct interpretation of relations is highly dependent on context as we discuss later on. The important point is that the chosen set of relations should be meaningful to the people using it and consistency in its use should be maintained.

TOOR is not a system for reasoning about requirements. It is rather directed to tracing them. However, some aid may be included. For example, a *Satisfaction* relation may be defined in TOOR and may be used in requirements tracing. The ultimate satisfaction of a given requirement is determined by some evaluation criteria, usually negotiated with the user. The satisfaction criteria is most desirably established by a test that may or may not be objective. Consider, for illustrative purposes, the definition of

```
omod REQ-SATISFATION is
  class Requirement .
  protecting SATISFTYPE .
  at satisf-test  : Requirement -> SatisfType [default: noteval] .
  at satisfaction : Requirement -> Bool .
```

```
var R : Requirement .
bax satisfaction(R) = true if satisf-test(R) = satisfied or
                              (satisf-test(R) = noteval and
                               is-der-satisfied(R) = true and
                               is-ref-satisfied(R) = true) .

endo
```

The `satisf-test` attribute may have the values of `satisfied` if the requirement has been satisfactorily subjected to the evaluation criteria, `notsatisfied` if the requirement has failed to pass the satisfaction criteria, and `noteval` if the requirement has not been tested with respect to the satisfaction criteria. This attribute is filled in by the user according to the outcome of the satisfaction criteria test.

A requirement is satisfied if it has satisfactorily passed the satisfaction criteria test, but not all requirements need to be tested. According to our definition of derivation we consider a requirement as satisfied if all requirements from which it is derived are satisfied. Also, a requirement is considered to be satisfied if all of its refining requirements are satisfied. This notion of satisfaction is defined by the built-in axiom for the `satisfaction` attribute. The function `is-der-satisfied` is a built-in function which given a requirement `R` returns true if all requirements from which `R` is derived are satisfied. The function `is-ref-satisfied` is defined similarly. The attribute `satisfaction` being defined by an axiom is a derived attribute and, therefore, the user does not have to type its value: it is always computed.

Satisfaction can then be used to trace requirements. The regular expression

```
[[Requirement x if satisfaction(x)]]
```

gives all satisfied requirements, and the regular expression

```
manualA Extract [[Requirement x if satisfaction(x)]]
```

gives all requirements extracted from manual `manualA` that have been satisfied. These kind of definitions may be used to trace inconsistencies. For example, every time a requirement `ReqA` fails to pass a satisfaction criteria test we can trace it back to see if it is derived from requirements that have been satisfied:

```
[[Requirement x if satisfaction(x)]] Derive ReqA
```

or, we can trace it forward to see if requirements refining it have been satisfied:

```
reqA Refine [[Requirement x if satisfaction(x)]]
```

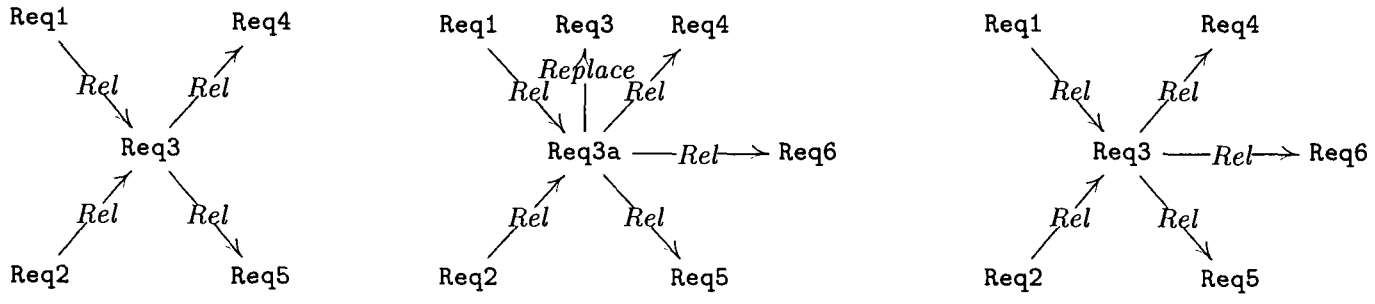


Figure 6.2: The requirement `Req3` in situation (A) is replaced by requirement `Req3a` which is later on related to requirement `Req6`. The final configuration is shown in (B). The user can see either configuration states (B) or (C). In particular, the user can restore the situation (A) before the replacement. In this case the relation between `Req3a` and `Req6` and the objects `Req3a` and `Req6` themselves are not shown because their creation dates are greater or equal to the creation date of the *Replace* relation.

6.2.2 Changing and Abandoning Requirements

Requirements can be changed in various ways. For example, an attribute may be changed if some mistake is discovered. The mistake may be corrected by simply editing the attribute value. In such a case, there is no record of the original attribute's content, although some other effect of the change may be registered. For example, suppose that some attribute of a requirement is edited to correct an error. The original content of the requirement's attribute is lost but a relation *Support* between the requirement and the person supporting (authorizing) the change may be registered. Alternatively, a requirement may be changed and the old version of it kept for traceability purposes.

6.2.2.1 Replacing Objects

The relation `TOORReplace` is defined in the standard project specification as a relation among objects of any class (`ToorObj`). It is used to indicate the replacement of an object by another whilst maintaining the replaced object in TOOR's database for future reference. The replaced object remains available for tracing. This kind of relation is useful if the change leading to the new version of the object is likely to be re-assessed later. In this case instead of just editing the object it is better to replace it and maintain the old version.

When an object is replaced by means of a `TOORReplace` relation, TOOR adjusts its configuration state by transferring all relations to and from the old object to the new one. The transference of relations from the replaced object to the object replacing it takes into account the configuration dates of the involved objects so that it is possible to have a sequence of replacements and restore any of the previous situations. Figure 6.2 explains this mechanism.

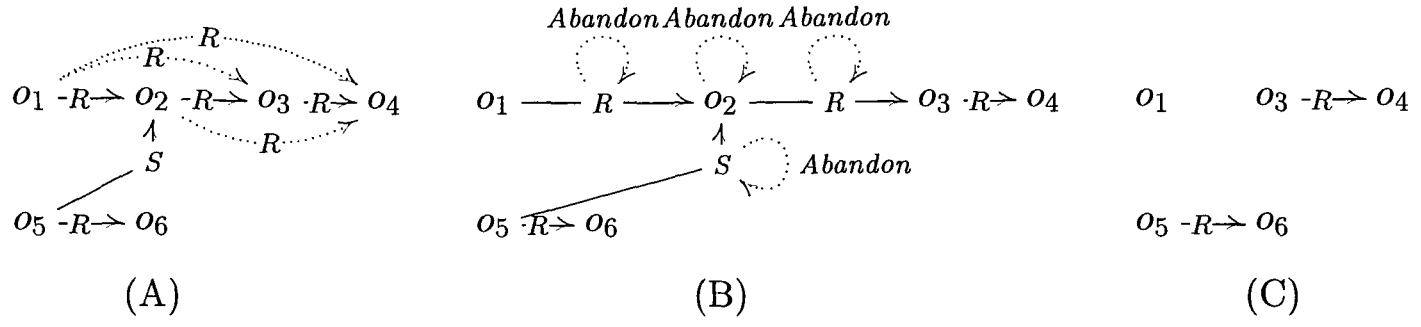


Figure 6.3: Consider objects $o_1, ..., o_6$ related as shown in (A). Assume that the type of R is transitive and that of S is ordinary. The thick arrows represent links made by the user, and the dotted ones those automatically computed by TOOR. When abandoning object o_2, the situation changes from (A) to (B). Note that only those relations made by the user are marked as abandoned (i.e., related by *Abandon*). The others can be recovered from the axioms. The user sees the graph (C), since abandoned objects are not normally shown.

The `TOORReplace` relation may be subclassed. This gives the flexibility of adapting the relation to particular situation, e.g., defining a replace relation which contains an attribute used to annotate the relation, e.g., giving a reason for the modification, a warning of dire consequences if the object is reinstated, etc. All subclasses of `TOORReplace` are treated in the same way as `TOORReplace` itself. Replaced objects do not appear in the configuration state, unless the user states otherwise by means of a specific command .

6.2.3 Abandoning Objects

Another kind of change occurs when an object is later found to be unnecessary, or inappropriate. One can either *delete* such an object, losing all trace of it, or one can mark it as *abandoned*, to preserve it for future reference. In terms of requirements, the first option is usually taken near the beginning of a project, when requirements are most unstable. Many requirements could have been developed tentatively, or hypothetically. The second option is described by a relation.

`TOORAbandon` is a relation class defined in the standard project specification with the constraint that both the source and the target objects of any of its instances should be the same. It is basically a relation from an object to itself. The object related in this way is hidden from most of the normal TOOR operations. The relations involving the abandoned object are also hidden and everything works as if the object was deleted. However, the user has the option of making the abandoned object visible again so that it may be used in tracing procedures. As with replacement, the `TOORAbandon` relation may be subclassed and the subclasses are treated by TOOR in the same way as `TOORAbandon` itself.

One question about deleting and abandoning requirements is how related objects are affected. As explained in Chapter 2, some links between objects are automatically generated

by axioms. TOOR can differentiate between links made by users and those generated by axioms. When deleting an object, links to the deleted object are also deleted, as are links generated from them. The same applies to abandoning an object: both its direct links and its computed links are automatically abandoned. In case of abandonment TOOR marks as abandoned all relations involving the abandoned object. The graphs in Figure 6.3 illustrate this situation.

6.3 Tracing Object Modifications

Objects in TOOR have their creation and modification dates as automatically maintained attributes. They may be used to set up baselines where the configuration state consists only of objects and relations created prior to a certain date. The traceability mechanisms of TOOR can then be applied to the previous configuration. Also, the creation and modification dates may be used to trace objects that have their state changed. Since at creation time both dates are set equal, the regular expression

```
[[ToorObj x if t$creation(x) =/= t$lastalter(x)]]
```

results in all objects that have been modified, and the regular expression

```
[[ToorObj x if t$creation(x) =/= t$lastalter(x) and
                t$lastalter(x) > basedate]]
```

results in all objects that have been modified after some date given by **basedate**. This gives a functionality similar to the Unix MAKE [43] system in accessing dependent files based on their modification dates.

The MAKE system embodies a very simple idea which is based on the dependencies among files, the establishment of target files, and on the association of commands to target files. For files `tar1`, `tar2`, `dep1`, `dep2`, a standard MAKE declaration has the form

```
tar1 tar2 ::  dep1 dep2
  command1
tar1 ::  dep3
  command2
```

where `tar1` and `tar2` are targets depending on `dep1` and `dep2`. The file `tar1` also depends on `dep3`. Every time a target file is younger (has a more recent modification date) than any of the files on which it depends the command associated to the declaration stating the

dependence between them is executed. For example, if `tar2` is younger than `dep1` or `dep2` the command `command1` is executed. If `tar1` is younger than `dep1` and `dep3` both commands `command1` and `command2` are executed.

In TOOR the dependence relation is asserted by the user creating the corresponding instance relating the two objects. For the situation above, the following TOOR module could be generated:

```
omod MAKE-EXAMPLE is
  tclasses Depend ToorObj .
  retain ToorObj : tar1 tar2 tar3 dep1 dep2 dep3 .
  retain Depend  : (tar1,dep1)  (tar1,dep2) (tar2,dep1) (tar2,dep2)
                   (tar1,dep3) .
endo
```

The module above is just a textual description of a dynamically created situation: the objects are registered through templates as well as relation instances involving them. Given an object `tar1`, for example, the regular expression

```
tar1 Depend [[ToorObj x if t$lastalter(x) < t$lastalter(tar1)]]
```

retrieves all objects `x` on which `tar1` depends and that are older than it. The automatic generation of such expressions are not part of the TOOR's environment and neither are automatic command executions based on them. On the other hand, by dealing with a variety of relations and by composing different regular expressions the user can trace objects in a finer grained way.

6.4 Discussion

Requirements evolution is a critical aspect of software development. The examples above show how such changes can be documented in TOOR, while preserving the original state of objects and relations. Appropriate support for evolution seems crucial to the success of any tracing system.

Notice that we did not suggest any particular model for software development. The relations discussed above could be part of such a model, but they are by no means definitive. The point of the project specification is to allow each culture, organization, and team of practitioners to support its own values and practices, by defining their own classes of objects and relations, and interpreting them in their own context.

Of course, some relations may have a very precise meaning, insofar as the meaning assigned to validate them is considered precise. This is the case of refinement for mathematical

specifications. In such a case, it is even possible to require a mathematical proof whenever one specification is registered as refining another. This can be done by creating special attributes in the corresponding relation class to hold the proof. We could even require that a proof checker accept the proof before the relation is registered. Such a practice would be considered valuable by some cultures and anathematized by others (see [54]).

The cultural interpretation of relations is more evident for those that do not have precise semantic definitions. For example, what does it mean for a requirement to be *Extracted* from a document? Is it that the document contains a statement of the requirement, or does it suffice to provide some evidence? If so, what degree and kind of evidence is sufficient? The same holds for other relations. For example, what does it mean for one natural language requirement to be *Derived* from another? Precise definitions can and should be sought, but the decision whether a particular case fits the definition is still a matter for interpretation.

Informality occurs in software development, and can even be helpful, particularly in the initial stages. If a given relation is considered too vague, one can choose not to register it in TOOR, but the concept behind the relation may nevertheless inform much of the practice of the development process. The question is which informal relations should be documented, and how much is manageable. Once a decision is made, TOOR can help by providing a database with automatic checks and annotations, and with convenient trace mechanisms.

Chapter 7

Tracing With Existing Domain Models

TOOR embodies a very general and flexible approach to traceability. However, it does not provide a model for the domain in which to perform the tracing. This is an important issue for an effective use of TOOR's features. Since the traceability model of TOOR is based on relations among objects, the design of the types of objects and their interconnections has to be carefully considered. A wrong choice may impair the development process as much as a meaningful design of the objects and relations to be used facilitates it. The project specification is suitable for defining the set of object and relation classes appropriate to each development situation according to the culture of a particular organization or development team. The project specification can also be used to incorporate into TOOR existing domain models specifying a particular set of objects and relations among them. Moreover, the ability to use existing domain models and to create specific models makes TOOR particularly suitable for requirements related research.

In this chapter we illustrate these points by showing how the project specification can be used to formalize two existing traceability models. We also discuss the gains and losses of such use.

7.1 Contributions Structures

Contribution Structures is a model that addresses the crux of the requirements traceability problem, which is regarded in [63] to be the difficulty in identifying the human sources of requirements, requirements related information, and requirements related work. It consists of a web of relations among contributors (the agents of a contribution) and the artifacts resulting from their contributions. It also encompasses relations among contributors themselves, revealing organizational structure, and relations among artifacts, revealing semantic depen-

dencies, as well as (derived) relations indicating notions such as social role, commitment, temporality, etc.

Here we present a simple account of how the contribution structures model can be formalized in a project specification for use in TOOR.

Agents are the human participants in the requirements engineering process. They may be modelled by a simple class `Agent` or by a class hierarchy showing their organizational structure. Artifacts are anything produced in the requirements engineering process that have a physical existence. They may include informal notes, recorded interviews, diagrams, etc., and they may be modelled as subclasses of a common superclass `Artifact`.

The basic kinds of contribution relations between agents and artifacts reflect the capacity in which the contribution is made. They are the *Principal* contribution relation, where the related agent is the one whose position is established by the information in the artifact, the *Author* contribution relation, where the agent is the person responsible for the content and structure of the artifact, and the *Documentator* contribution relation, where the agent is responsible for the physical and documentational aspects of the artifact. Below is the specification for the *Principal* relation. The others are specified in a similar way.

```
omod PRINCIPAL is
  extending RELATION[AGENT,ARTIFACT] * (class Relation to Principal) .
endo
```

The contribution relations may be qualified to show details of the contribution. In TOOR this is specified by appropriate attributes for each relation class. Below is an example of such a qualification.

```
omod PRINCIPAL is
  extending RELATION[AGENT,ARTIFACT] * (class Relation to Principal) .
  protecting CONTRIB-TYPE .
  at contrib-type : Principal -> PcontribType .
endo
```

The module `CONTRIB-TYPE` would contain the specification for the datatype `PcontribType` which would allow the values of `Approved`, `PendingApproval`, and `NotApproved`.

Further relations can be derived based on the various capacities exercised by an agent when contributing to the production of an artifact. The *Role* relations are representative of this kind of derived relations. They reflect the social roles actually played by agents when producing artifacts and they are determined by the capacity relations that an agent takes part in. Figure 7.1 shows a set of these relations and the code below gives the specification for the *Devisor* relation.

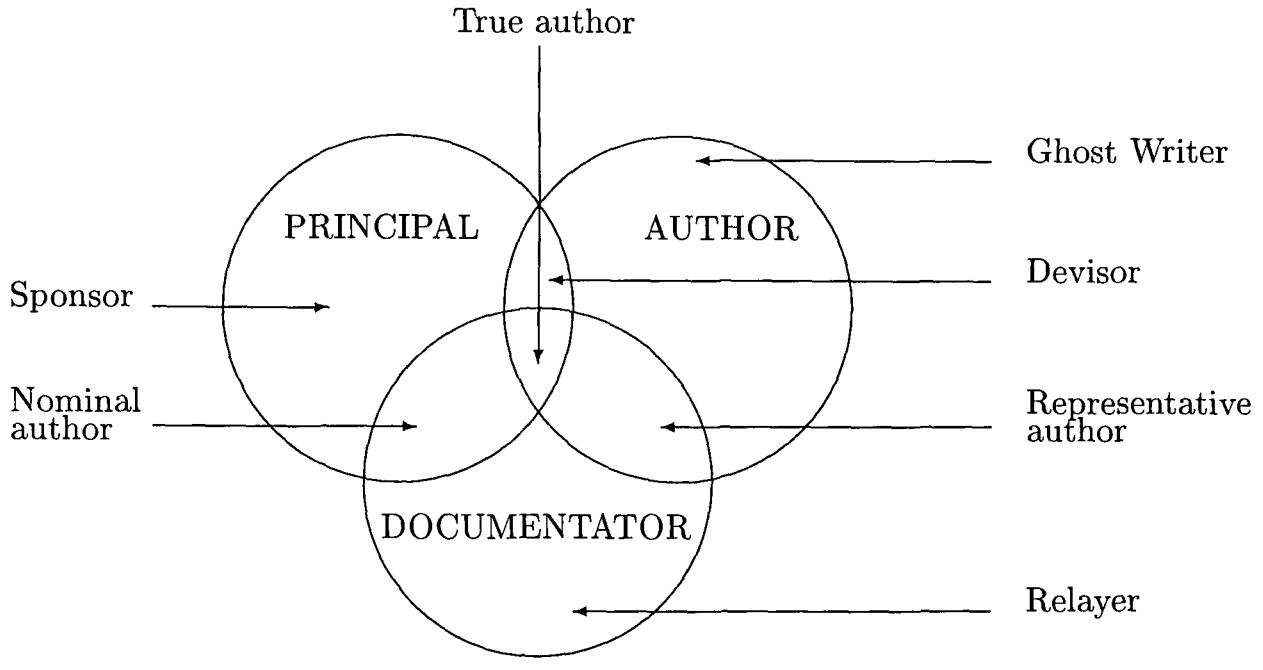


Figure 7.1: Role Relations in the Contribution Structures Model

```
omod DEVISOR is
  extending RELATION[AGENT,ARTIFACT] * (class Relation to Devisor) .
  me is-Devisor : Devisor -> Bool .
  var A : Agent .
  var D : Artifact .
  ax is-Devisor(A,D) = is-related('Author,A,D) and
                       is-related('Principal,A,D) .
endo
```

With the above specification the `Devisor` relation does not need to be directly registered by the user. An agent `A` is considered by TOOR to be the devisor of an artifact `D` whenever `A` is related to `D` by an `Author` relation and by a `Principal` relation. This agrees with the rationale behind the production of the role relations.

Other relations are specified in a similar way but now we turn our attention to the use of the Contribution Structures model to trace requirements. Contribution Structures address what is called in [63] personnel based traceability, i.e, questions relative to the human resources of requirements. Gotel [63] has empirically identified some frequently asked questions related to this kind of traceability. We give the regular expression form to answer some of them once the Contribution Structures model is specified in TOOR.

For the question of "Who has been involved in the production of this requirement and how", the regular expression

```
[Agent] ToorRel reqA
```

would result in the configuration state showing all chains of relations from objects of class

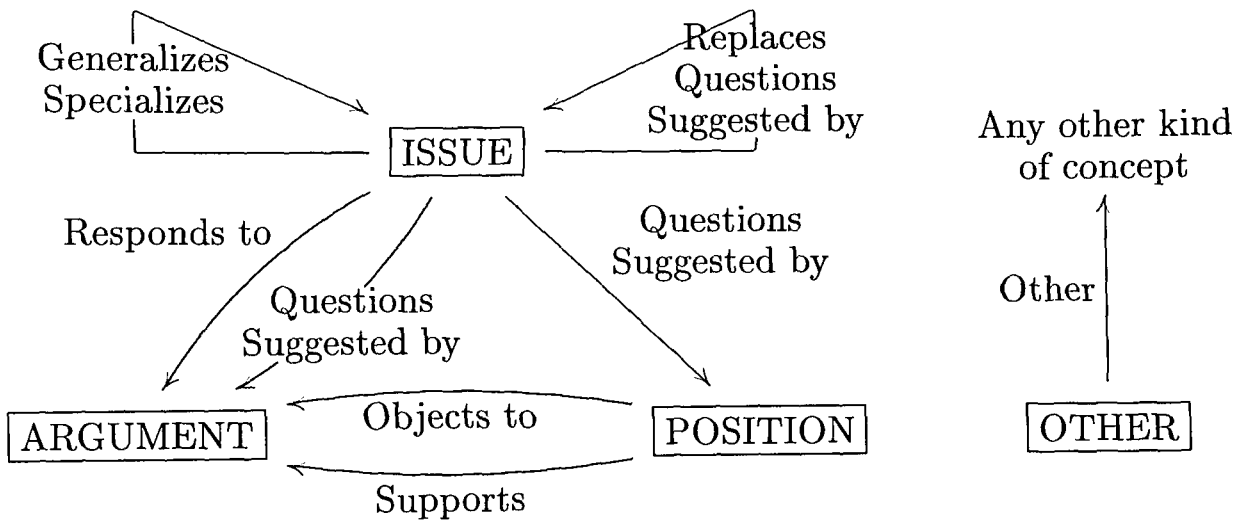


Figure 7.2: IBIS Model Concepts and Links

`Agent` to requirement `reqA`. This would show the contribution relations involving the agents related to `reqA` but also any other relation in which the agents take part. If we want to restrict the result only to the contribution relations we could express the following pattern

```
[Agent] (Principal | Author | Documentator) reqA
```

This would result in the principal, author, and documentator contributions to `reqA`. The same can be done to express other kinds of patterns. For the question of "Who was originally responsible for this requirement", the regular expression

```
[Agent] Principal (; ToorRel)* reqA
```

would give the agents responsible for the positions established in requirement `reqA`. This particular form retrieves agents related to `reqA` by the `Principal` relation, and also agents related by the `Pricipal` relation to some artifact which is in turn related in some way to `reqA`. The relation class may be modified if other kinds of responsibility are desired.

7.2 gIBIS-like Models

The gIBIS tool [25, 124] is designed to capture design rationale by providing automated support for discussion and negotiation of design issues. Basically, it implements the *Issue Based Information Systems* (IBIS) model which consists of four concepts and nine kinds of links to relate them.

The basic concept of the IBIS model is that of *Issue*. An Issue may be any problem, concern, or question that may require discussion (and resolution). Each Issue may have many *Positions* ascertaining a possible solution for it. Each Position may in turn have *Arguments* supporting or rejecting it. The links used to relate these concepts are as follows:

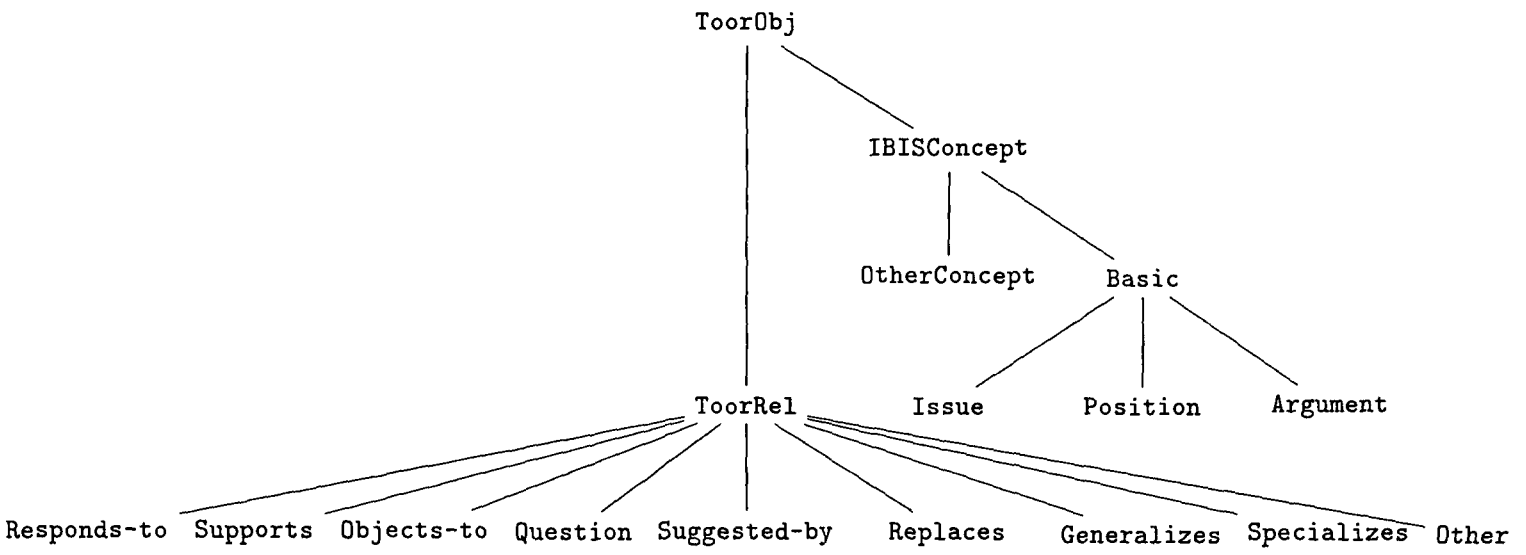


Figure 7.3: TOOR Class Hierarchy for the IBIS Model

the *Supports* and *Objects-to* links relate an Argument to a Position. The *Responds-to* link connects a Position to an Issue. An Issue may be linked to other Issues by a *Generalize* or *Specialize* links. Issues are also linked to other Issues, Positions, and Arguments using the *Question* or *Suggested-by* links. The *Other* concept is an escape mechanism. A concept of type Other may be linked to any other concept by means of the *Other* link. The overall structure of the model is shown in Figure 7.2. We note that the links Question and Suggested-by, as well as Other, may involve different kinds of concepts. The table below summarizes the IBIS model in terms of each possible application of its links.

Links	Concepts		
	from		to
Support :	Position	→	Argument
Objects-to :	Position	→	Argument
Responds-to :	Argument	→	Issue
Generalizes :	Issue	→	Issue
Specialize :	Issue	→	Issue
Replaces :	Issue	→	Issue
Question :	Issue	→	Issue
Question :	Issue	→	Position
Question :	Issue	→	Argument
Suggested-by :	Issue	→	Issue
Suggested-by :	Issue	→	Position
Suggested-by :	Issue	→	Argument
Other :	Other	→	Other
Other :	Other	→	Issue
Other :	Other	→	Position
Other :	Other	→	Argument

To use the IBIS model in TOOR we just specify a class for each concept according to the hierarchy shown in Figure 7.3. The links are specified as relations among the appropriate classes. Those links that may link concepts of different classes are defined as relations having

the source and target attributes taking values from objects of a common superclass. For example, the links Question, Suggested-by, and Other are specified by

```
omod QUESTION is
  extending RELATION[ISSUE,BASIC] * (class Relation to Question) .
endo

omod SUGGESTEDBY is
  extending RELATION[ISSUE,BASIC] * (class Relation to Suggested-by) .
endo

omod OTHER-LINK is
  extending RELATION[OTHER,IBISCONCEPT] * (class Relation to Other) .
endo
```

Observe that the use of additional classes as superclasses of the basic concepts does not alter the IBIS model. Those superclasses are not intended to have objects of their own and they may be suppressed from the TOOR menus. Nevertheless, even if some classes are not available to the user the class hierarchy is internally preserved as a whole and any object of class Issue, Position, or Argument is also considered to be an object of class Basic. Therefore, a relation Suggested-by may be made between objects of class Issue and objects of classes Issue, Position, or Argument, just as required by the IBIS model.

Once a project Specification for the IBIS model is ready a project may start and the traceability facilities of TOOR may be used to retrieve parts of the IBIS structure. As reported in [25] the search mechanisms of the original IBIS are those provided by the hypertext browsing and queries by example based on the contents of nodes. Within TOOR we can refer directly to the existing relations. For example, wishing to get a picture of all discussions that has been initiated as a sub-product of an original set of issues, we may express the query "All Issues suggested by Positions or Arguments" as a regular expression

```
[Issue] Suggested-by [Position] | [Issue] Suggested-by [Argument]
```

We may also experiment with extensions to the model. For example, we may verify the effect of considering the Specialize relation transitive by defining it as

```
omod SPECIALIZE is
  extending RELATION[ISSUE,ISSUE] * (class Relation to Specialize) .
  var S : Specialize .
  ax t$type(S) = transitive .
endo
```

This makes an Issue specialized by an Issue B which is in turn specialized by an Issue A to be also an Issue specialized by A. This may or may not make sense. The point is that

with only indirect links being created by virtue of the axiom for transitivity, after having tried the transitive version of Specialize the user may discard it by removing the axiom and reloading the project. Nothing of the original state is lost, i.e., all (direct) links created by the user are preserved. Other possibilities may be tried. The following code uses an indirect relation axiom to specify the transitivity of Suggested-by with respect to Issue concepts.

```
omod SUGGESTEDBY is
  extending RELATION[ISSUE,BASIC] * (class Relation to Suggested-by) .
  me is-Suggested-by: Issue Issue -> Bool .
  vars I1 I2 : Issue .
  cax is-Suggested-by(I1,I2) = member(image('Suggested-by,I1),I2) or
    inter(image('Suggested-by,I1),image-1('Suggested-by,I2) =/= empty .
endo
```

An important aspect of TOOR is its capability to be used throughout the development. Therefore, a model like IBIS may be enlarged with classes to define artifacts and relations for latter phases of system development. We illustrate this feature with the REMAP model.

7.2.1 Enlarging Models of Objects and Relations

The REMAP [118, 119, 120, 121] model incorporates the IBIS model. It adds to the original set of IBIS concepts (Issue, Position, and Argument), the new concepts of *Assumption*, *Decision*, *Requirement*, *Constraint*, and *DesignObject*. The links among IBIS concepts remain the same and new ones are created involving the new concepts. Some examples of these new links are: a Requirement *Generates* an Issue, an Assumption *Qualifies* an Argument, an Argument *Depends-on* an Assumption, and a Decision *Resolves* an Issue.

A project specification may be created for this extended version in exactly the same way we did for the IBIS model: object classes are specified for each concept and relation classes are specified for each kind of link. Traceability is then available throughout the project. This, for some models, may be an enhancement in itself. In the case of REMAP the system is intended to be connected to the prototyping system CAPS [89, 90, 87] where design artifacts have a simple kind of link relating them to a specific requirement. Traceability in this context goes from the design artifact to the requirement linked to it and from the requirement to the design rationale captured by the extended IBIS model. The reverse direction is also possible but again in two steps, with the requirement being always the connection element between design rationales and CAPS design artifacts. Within TOOR the user may retrieve directly parts of the design rationale structure related to a given design object. A regular expression like

```
[Assumption] ToorRel designobjA
```

would retrieve all Assumption objects related to the design object `designobjA`. It would show in terms of a configuration state all relations forming the existing chains from Assumption to Arguments to Positions to Issues to Requirements to object `designobjA`. Moreover, if FOOPS objects are used as the design objects of the model all automatic generation of traces described in Section 3.5 would be available.

The formal specification of relations may be used to implement in TOOR additional functionality. Originally the IBIS model is not a resolution model. Issues are always open to further assessments. The REMAP model incorporates a resolution mechanism through the Decision concept. A Decision is something that Resolves an Issue by selecting a Position related to that Issue. Once a decision is made on some issue the next step is to *Imply* or *Generate* a design *Constraint*. If one wants to stop the process of discussing an issue after a decision has been made, it is necessary to prohibit the linking of new issues, positions, and arguments to the already decided issue. This is done in TOOR by specifying for each relation involving an Issue the necessary constraint. We exemplify with the relation *Generalizes*:

```
omod GENERALIZE is
  extending RELATION[ISSUE,ISSUE] * (class Relation to Generalizes) .
  vars I1 I2 : Issue .
  cax may-Generalizes(I1,I2) = true if image-1('Resolves,I1) == empty .
endo
```

As explained in Section 2.4.8, for every relation class `R` the axiom `may-R`, if provided in the project specification, is taken into account by TOOR when evaluating the creation of objects of class `R`. In this case an object of class `Generalizes` may only be created if its source object (`I1`) is not resolved, i.e., if it is not being used as a target object of a relation object of class `Resolves`. This is expressed by the conditional axiom saying that the counter-image of `I1` under `Resolves` should be empty.

Of course, it is not feasible to definitively close an Issue after a decision has been made. After all, the changing nature of the development process may always require the reassessment of previous 'decided' issues. A solution accounting for this fact would be to specify an attribute `status` for the class Issue taking one of two possible values: `open` and `closed`. An open issue would be the subject of further discussion. The axioms would then be expressed as

```
omod GENERALIZE is
  extending RELATION[ISSUE,ISSUE] * (class Relation to Generalizes) .
  vars I1 I2 : Issue .
  cax may-Generalizes(I1,I2) = true if status(I1) == open .
endo
```

The attribute `status` may also be specified for the relation class Resolves instead of for the class Issue. The attribute could have a default value of `closed` meaning that every time a decision is made on an issue, the issue is closed to further discussions. To reopen an issue would require the relation object resolving it to have its `status` attribute modified from `closed` to `open`. The axiom for this kind of solution would be

```
omod GENERALIZE is
  extending RELATION[ISSUE,ISSUE] * (class Relation to Generalizes) .
  protecting RESOLVE .
  vars I1 I2 : Issue .
  cax may-Generalizes(I1,I2) = true
                              if is-open(all-Resolves,I1) == true .
endo
```

In this solution, a function `is-open` should be defined in the project specification to return `true` if the value of `status` is `open` for all objects of class `Resolves` having `I1` as its target attribute.

7.3 Keeping Specific Features

Although efficient for traceability purposes TOOR cannot incorporate every domain model without losing some of its functionality, and it certainly cannot replace dedicated tools designed to explore these domains. For example, the gIBIS tool is specifically designed to capture, in a hypertext-like way, the dynamics of multi-part discussions over a large set of issues. It has features, like the collapsing of entire sub-networks into a single node, that greatly enhances the ability to visually analyse large networks. This feature cannot be replicated in TOOR.

The KJ [134, 101] editor is another example of losing functionality when translating a model of objects and relations into TOOR. The KJ method is designed to facilitate and improve the organization and generation of ideas. Basically, a person using the method places small cards around a large surface. The cards are grouped and related to each other according to semantic dependencies which depend on the problem at hand and on the person carrying out the method. The structure formed in this way is said to facilitate the organization of ideas and the generation of new ones. The KJ method has found applications in many areas of the Japanese business community and has been proposed for use in requirements analysis [134].

The translation of the KJ method into a model of object and relation classes is fairly simple. There are two kinds of classes for objects, the card and the group, and there are few relations among them. Using the KJ model as part of a larger project structure in TOOR brings the benefit of keeping in a single environment the requirements analysis, through the

KJ method, and the subsequent use of requirements for the later development phases. The benefits may be increased if the card objects were defined to contain hypermedia attributes such as images and sounds. However, one central aspect of applying the KJ method is the ease with which the cards may be moved around the surface and the ability to control the visual appearance of the overall structure. To this end, the KJ editor interface has specific facilities for dragging and dropping and for defining the shape of the relation lines and group boundaries. These facilities are lost when the method is used in TOOR.

Nevertheless, the possibility of translating domain models into a structure of objects and relations in TOOR has the advantage of making traceability available in a uniform way to the whole spectrum of the development process. It also has the advantage that in TOOR we can try alternative variants of the incorporated model. If it is important to keep all the functionality of some tool for a given method or domain, the translation process may be done in a three-step procedure:

- **step 1** Use the method X with a dedicated tool for X.
  Traceability in this phase will be restricted to the mechanisms available in the tool for X and will be limited to the objects in the domain of X.
- **step 2** Translate the X model into TOOR and proceed with the development.
  The traceability model of TOOR is then applied to objects and relations of the project.
- **step 3** If objects coming from the domain of X need to be re-assessed, translate the corresponding part of the TOOR structure back into X.

This procedure works if the tool for a given method X can export and import information about the X domain. Specific translators are likely to be needed.

7.4 Discussion

The use of TOOR presupposes a configuration of objects and relations to model the domain where the tracing is to be done. The project specification provides a flexible way to define domain models containing the relevant classes for objects and relations that are adequate for particular situations and particular development teams. Nevertheless, this generic approach only works if it can be demonstrated that it can reasonably encompass different ways of viewing objects and their relationships.

In this chapter we applied TOOR to two traceability models that we believe provide useful modeling concepts regarding the objects and relations relevant for requirements tracing. This demonstrates the feasibility of defining different project environments in a profitable way.

By using an existing model to base the definition of classes for objects and relations in TOOR a development team can preserve the vocabulary and concepts familiar to them. Moreover, in many cases TOOR may be used to increase the traceability efficiency of the model being used. At least, it enhances traceability by providing uniform traceability features throughout system development. Of course, not all functionality of existing models and tools may be preserved and some compromise has to be made if TOOR is to be used incorporating other traceability models.

An important related point demonstrated in this chapter is the feasibility of using TOOR for requirements research. It may be used as a prototyping tool providing a quick way to develop environments in which to test (parts of) some proposed model, and then develop case studies and verify alternative approaches.

Chapter 8

Comparisons

In this chapter we give a summary of the features of our model and compare it with other models and approaches. The comparison is divided in two sections: one for academic approaches and the other for industrial strength.

8.1 TOOR summary

We have presented a traceability model and its associated tool. The model is based on the use of relations to represent traces of objects. However, it is not tied to any particular representation for the objects and relations. It is instead based on the formalization of configurations of objects and relations as a graph and on the definition of patterns of related objects that may be extracted using regular expressions. Relations are implemented as objects so the features of the model apply to objects and relations in a uniform way. The traceability model also addresses other features necessary for an efficient use of tracing. First, it is user-definable with respect to the traces that may be registered and also with respect to the objects that may be traced. Second, the model recognizes that the tracing should be incorporated into the development process and that some traces may be automatically produced. To this end the model incorporates a modularization mechanism, which is also user-definable, and it is integrated with FOOPS as the specification and coding language for a system being developed in a project. The model is particularly suited for an object-oriented development process using FOOPS as the specification and coding language for the system being built. The use of FOOPS allows the automatic production of traces involving the classes, attributes, and methods used to define the system, and automatic checks in case these traces cannot be automatically produced. Finally, the model recognizes the need for varied ways of tracing and for an adaptable environment in which the tracing is carried out. It possesses several browsing mechanisms and hypermedia features. All these features are integrated. A regular expression may refer, through FOOPS axioms, to the contents of external files, modules may

Trace Definition

Traces are defined through a formal specification of objects and relations allowing

- Definition of object structure.
- Definition of derived attributes that can be used to produce automatic values for attributes.
- Definition of object methods that can be used as part of tracing expressions, e.g., to search external files.
- Definition of relations by axioms.

Trace Extraction

Traces are extracted through different procedures allowing

- Selective tracing through use of regular expressions
- Interactive tracing through use of project's module structure
- Free tracing through browsing the configuration of objects and relations
- Trace of information stored in external files
- Access to information stored in a variety of formats

Trace Production

Traces are produced using a tool with the following characteristics

- Definition of modules providing scope for object creation and object use.
- Use of templates to create objects and relations
- Use of external files as objects' attributes values
- Automatic creation of relations based on relation axioms
- Automatic creation of relations based on the syntax of FOOPS objects
- Automatic creation of relations based on the object class hierarchy
- Automatic creation of relations based on the project module structure
- Modification of the tracing environment at the project specification level and at the interface/template level

Regular Expressions

Regular expressions specify patterns of objects and relations using

• Object identifiers	matched by objects with identical identifiers.
• Regular expressions over the identifier alphabet	matched by objects whose identifiers match the regular expressions.
• Class identifiers	matched by objects whose classes are subclasses of the given one.
• Relation class identifiers	matched by relation instances whose classes are subclasses of the given one.
• FOOPS axioms	matched by objects for which the given axiom evaluates to `true`.

Table 8.1: TOOR's Traceability Model Features

be used to restrict the search space for browsing and tracing with regular expressions, and browsing may be performed on objects resulting from tracing with regular expressions and on objects resulting from tracing through the module structure. These features are summarized in Figure 8.1.

8.2 Academic Approaches

Research on requirements has produced many languages, tools, and models dealing with several aspects of requirements, including informal and formal specification of requirements; requirements analysis and validation techniques; requirements domain models; development environments encompassing the requirements phase; etc. Nearly all of these provide some traceability mechanism. Some as a by-product of being able to generate reports and cross-references and others by explicit tracing features. Examples of the former are the requirements language processor RLP [30, 27, 32, 26] (more specifically the languages generated by it such as RTRL [135]) which can produce cross-reference tables, and PSL/PSA [136, 138] which provide an extensive documentation based on the specified requirements. An example of the latter is the requirements specification language RSL (part of the SREM methodology [4, 5, 6, 7, 8, 15]) which has the statements `traces-from` and `traces-to`. Nevertheless, we review, for purposes of comparison, only tools and models primarily designed to manage requirements traceability and of those we select only a few that represent some of the aspects we are interested in. We compare TOOR with

1. traditional approaches to requirements traceability, of which we choose ARTS [38, 47] as an early proposal and RADIX [150] as a more recent proposal;

2. approaches addressing pre-requirements specification traceability, of which we choose REMAP [119, 118, 121] for its orientation towards design rationale capture, RETH [79] for its hypertext capabilities, and Contribution Structures (CS) [63, 65] as a novel approach in this area;

3. approaches managing traceability across several development phases, of which we choose ESE [117] and SODOS [71, 72] for their configuration management facilities; and

4. approaches integrated with support for software development, of which we choose NATURE [77, 78, 110] for its process-oriented features.

Of course the chosen representatives for the approaches above greatly differ from each other. Some are well established models with extensive tool support, e.g., ARTS and NATURE,

others embody models that may be applied to a variety of development methods but whose current tool support is restricted to particular methods of system development, e.g., REMAP with respect to prototype development, and some are models whose current tool support are more 'proof of concept' than effective tools, e.g., CS.

8.2.1 Trace Definition

The relevance of trace definition refers to the adaptation of the model to different situations. In general, the more flexible and comprehensive the facilities to define what the traces and trace units are, the more adaptable a model is with respect to its application to different contexts. Trace definition regards the types of objects that can be traced and how they relate to each other. The word 'object' is used in a broad sense meaning any trace unit covered by the model. Generally, we refer to them as trace units but we use the concepts of the model when talking about particular models. We consider that traces are defined by any reference from one trace unit to another. This includes explicit mechanisms like relations and links but also references made through the content of an object's attribute, for example. We analyse three aspects of trace definition. For a given model, its **definition mode** is said to be *user-defined* if it allows the user to define traces and trace units, *pre-defined* if traces and trace units are fixed, and *mixed* if it has both features. The **definition range** is said to be *pre-RS*, if the traces and trace units are intended to represent traces related to pre-requirements specification traceability, and *post-RS* if they are intended to represent traces related to post-requirement specification traceability. In case of *post-RS* the classification may have appended the particular life-cycle phases covered by the model. For example, post-RS[Requirements,Design] indicates that the model is intended for post-RS traceability covering the requirements and design phases. The use of post-RS alone indicates that the model is intended to cover all development phases. The **life-cycle definition** may be classified as *independent* if the model does not use life-cycle concepts for tracing, *dependent* if the model is designed for a particular life-cycle strategy, and *adaptable* if the model may be used under different life-cycle strategies. An independent model with respect to life-cycle definition may also be used under different life-cycle strategies but it does not define life-cycle structures and therefore it cannot use life-cycle concepts for tracing. Table 8.2 summarizes the result of classifying the traceability models according to these categories. The list below gives additional information.

ARTS Trace units are database records. Each record has an obligatory `derive` attribute which defines the basic tree-structure for organizing requirements. Traces are given by attributes referencing other trace units.

Model	Mode	Range	Life-Cycle
ARTS	pre-defined	post-RS[Requirements,Design]	independent
CS	mixed	pre-RS	independent
ESE	mixed	post-RS	independent
NATURE	mixed	pre-RS post-RS[Requirements,Design]	independent
RADIX	user-defined	post-RS[Requirements,Design,Test]	dependent
REMAP	pre-defined	pre-RS post-RS[Requirements,Design]	dependent
RETH	user-defined	pre-RS post-RS[Requirements,Design]	independent
SODOS	mixed	post-RS	adaptable
TOOR	mixed	pre-RS post-RS	adaptable

Table 8.2: Trace Definition of Traceability Models

CS The basic trace units are requirements, requirements related artifacts and people contributing to their production. Other concepts such as organizational structure may also be defined as trace units. Traces are defined as relations among trace units. They are user definable and some are indirectly defined in terms of previous ones.

ESE Trace units are files whose identifiers are registered in a relational database with user-definable attributes. Also groups of files may be registered as an object (trace unit) and may have user-definable attributes. Traces are defined by links declared between trace units and also by references determined by attribute's content. The links may be generic or specific depending on the type of information used to support their existence. Links are not user-definable.

NATURE Trace units are defined in terms of objects of given classes. There are specific classes for each process specified in the model. There are classes for rationale concepts, design concepts through Structured Analysis and Entity-Relationship methods, etc. Although comprehensive (230 classes) once the model is defined the user cannot add additional classes. Traces are defined as dependencies among the various concepts. The tool may be customized to include additional dependencies.

RADIX Trace units are marked parts of texts inside documents. There are a pre-defined number of marks to delimit parts of the text specifying their nature, e.g., require-

ment, explanation, keyword. Traces are defined by explicitly marking a trace unit with references to other texts and documents.

REMAP The trace units are the concepts of the IBIS model enlarged by additional concepts related to requirements rationale capture such as `Requirement`, `Assumption`, and `Decision`. At the design level the trace units are specification texts written in PSDL. Traces are defined as links between trace units. The only link from a requirement to a specification is established by the PSDL statement `by requirement`.

RETH Trace units are hypertext nodes defined as frames which may be user-defined in terms of their structure, i.e., type and content of their slots. Traces are links between hypertext nodes but they also are implemented as frames. There is also an automatic definition of traces through inheritance provided by the frame system.

SODOS Trace units are documents with a pre-defined structure consisting of chapter, section, paragraph, and figure hierarchies. Although it is possible to have graphics and figures as part of a document they should be stored in the same database: there is no reference to external files. Traces are pre-defined relations between documents. There are two types of relations: intradocument relations, used to relate components of a document to other components of the same document, and interdocument relations, used to relate components of a document to components of other documents.

8.2.2 Trace Production

The relevance of trace production refers to the accuracy of the model with respect to the registration/capture of its defined traces and trace units. In general, the more automatic the way traces and trace units are registered, the more accurate the model is. Of course, this is a concept that applies to the tool supporting the model rather than to the model itself. Trace production concerns the way traces and objects are registered. For traceability models like CS and SODOS we go beyond the prototype implementation and consider, for example, the theoretical possibility of trace information being automatically extracted from documents. We analyse two aspects of producing traces and trace information. The **production mode** classifies trace production as *manual* if traces and trace units are manually registered, *off-line automatic* if they are automatically registered through a specific procedure, and *on-line automatic* if they are automatically registered as a result of registering related information

model	mode			reference	
	off-line		on-line		
	manual	automatic		indirect	direct
ARTS	√			√	
CS	√	√		√	√
ESE	√	√			√
NATURE	√	√	√		√
RADIX	√				√
REMAP	√	√		√	√
RETH	√	√		√	√
SODOS	√	√		√	√
TOOR	√	√	√	√	√

Table 8.3: Trace Production of Traceability Models

into the model. The off-line automatic production of trace information can be exemplified by a procedure to extract trace information from a document. In this case the document already contains the trace information to be registered but the actual registration does not occur as a result of registering the document; an external, specific procedure has to be performed to extract them. The on-line automatic production of trace information can be exemplified by the automatic registration of a relation based on an object's attribute. In this case the relation is registered as a result of registering the object. Trace production is also categorized with respect to its **development reference**. It is said to be *direct* if it uses as trace information the actual artifacts produced during development. It is said to be *indirect* if it refers to descriptions of artifacts, rather than to the artifacts themselves. Table 8.3 summarizes the result of classifying the traceability models according to these categories. The list below gives additional information.

ARTS Traces are produced as registers are entered and their attributes filled in.

CS Traces are produced as objects and relations are registered and are therefore manual. However, trace units may be automatically extracted from documents and there is also automatic production of traces given by indirect relations.

ESE Traces are created as they are registered into the database. The system supports version control of objects. For some type of objects, like computer programs, the necessary information to register them and their links can be automatically extracted.

NATURE Traces are automatically produced as a result of using specific editors to manipulate concepts. For example, creating an object through a particular editor may automatically produce traces according to the defined dependencies among objects. This happens even if the objects are manipulated by different editors.

RADIX The production of traces is through document edition to insert the necessary marks delimiting and referencing trace units.

REMAP Traces are produced as the objects and relations are registered in the model. There is no automatic production of traces.

RETH Traces are produced as nodes are entered and linked using the hypertext editor. There is a possibility for a partial automatic linkage of nodes based on their textual content.

SODOS Most of the traces are produced manually as documents as their relations are entered into the system. Some intradocument relations are produced automatically as a result of registering components of documents, e.g., adding a new section to a document automatically produces a `structure` relation between the document and the section.

8.2.3 Trace Extraction

The relevance of trace extraction lies in the flexibility with which trace information can be used. In general, the more varied the ways of extracting and presenting tracing information, the more flexible a model is in providing answers to traceability questions. Again, this feature is more concerned with the tool implementing the model than with the model itself. The **extraction mode** is classified as *browse*, if the user can interactively browse the information contained in the model, *query* if trace information may be extracted using queries, and *command* if trace information may be extracted by means of specific commands. Although queries may be issued as commands, we classify a model as having a query extraction mode if the user can formulate his own queries, and we say that a model has a command extraction mode if the procedures to extract trace information are pre-defined with the user just choosing the desired one. This is the case, for example, of commands to generate specific reports. The **extraction presentation** is used to classify a model as *textual* if the information is presented in a textual way, *graphic* if the information may be presented in a graphical way, and *hyper* if the information may be presented in a hypermedia fashion. The **extraction space** is used to classify models by the facilities they offer for limiting the search space for trace extraction.

model	mode			presentation	space
	browse	query	command		
ARTS			√	textual	fixed
CS	√	√	√	hyper	fixed
ESE	√	√		textual	fixed
NATURE	√	√		hyper	fixed
RADIX			√	textual	fixed
REMAP	√	√		graphic	fixed
RETH	√			hyper	fixed
SODOS	√	√		graphic	fixed
TOOR	√	√	√	hyper	variable

Table 8.4: Trace Extraction of Traceability Models

The space is said to be *variable* if it can be constrained by the user and *fixed* otherwise. The user can limit the extraction space using queries or commands that consider only certain information but this is not what is meant by a variable extraction space. The extraction space is said to be variable only if the user can determine it independently of any query or command. Therefore, in a variable space one query may produce different results. Table 8.4 summarizes the result of classifying the traceability models according to these categories. The list below gives additional information.

ARTS Traces are extracted by means of commands producing several forms of reports based on registers' contents.

CS Traces are extracted in multiple ways. Extraction occurs through traversing related objects. It seems that traces may be extracted by specific procedures to do the traversal, which characterize a command extraction mode, and by a generic traversing algorithm which characterize a query extraction mode. Also, the prototype implementation allows for a hypertext browsing of some objects (artifacts produced as marked up documents).

ESE Traces are extracted by queries to the database based on objects' attributes and on their links.

NATURE Traces are extracted using both a dedicated browser having access to all objects in the database or specific browsers for each process, e.g., hypertext and entity-relationship browsers. The objects may be retrieved in several ways.

RADIX Traces are extracted by producing reports based on marked documents.

REMAP Traces are extracted in multiple ways. The use of hypertext and the use of a reasoning maintenance system are two of them. Particularly the reasoning maintenance system helps in extracting traces in several different ways based on the degree of belief associated to links and in the establishment of different dependence relations based on the existence on actual links between objects.

RETH Traces are extracted by browsing the hypertext structure.

SODOS Traces are extracted by queries (query-by-example) retrieving information based on documents' contents and their relations.

8.3 Industrial Strength

There are many industrial tools currently available to trace requirements. Not surprisingly they all have common points either in the facilities provided to their users or in the techniques employed for tracing. First, they tend to favour established advanced techniques instead of new but unproven, or little experimented, ones. Second, to the extent they are market driven, they tend to be designed in order to use and communicate with other widely adopted models and tools.

Requirements traceability in these tools is usually available at all phases of the development from requirements to code. However, the links to design and code are usually made through interfaces to specific CASE tools. Although this is not a problem in itself, at the moment tool integration is still a complicated topic, accomplished most of the time by using external files as a bridge between tools [91]. Therefore, traceability is broken at interface points since requirements traceability tools usually trace only what is registered in their own databases. Tracing is based on established techniques like matrices, sequences of matrices, requirements tagging, database query languages, etc.

Nevertheless, comparison with industrial tools provides a good way to assess the current state of the practice. Also, because of their usually broad scope they offer a good basis for evaluating the application of requirements traceability through all phases of system development. In our discussion we use RDD-100 [91, 107, 142, 92], RTM [96, 83, 84, 144], and DOORS [115, 113, 114, 132] as representatives of the industrial strength in requirements management (requirements traceability, in particular). They are currently among the most popular and have already been described and compared a number of times [81, 63, 74, 98]. The features that distinguish them from TOOR and TOOR's limitations are also presented.

8.3.1 RDD-100

RDD-100, RDD-100 for short, emphasizes the modeling of the development process [91, 107, 142]. Its concepts are based on the use of different models to describe processes at different levels. These models are integrated under a uniform description. The mapping of elements from one model to another is automatic and transparent to the user through the uniform description characterizing each model. Each model gives rise to a process template through which model elements are entered into a System Description Database (SDD). Both the models and the templates are user-defined. For a traditional development structure there may be defined, for example, models for requirements, design, and implementation. The implementation model may contain elements for programs, data, and control structure of programs.

Traceability is achieved by links defined between model elements. Not only elements in the same model may be connected but also elements from different models, indicating the transformation process from one model to another, i.e., the flowdown of information. Once a set of models is defined the traceability structure is fixed, encoded in the relationships specified in the model's description. To change it, it is necessary to redefine the models with a consequent transfer of information from the original set of models to the new changed set of models. Traceability is maintained by editing information in the SDD via templates and specific windows. There are automatic facilities to extract information from external documents and files. The use of external documents is via macros defined for specific desktop publishing (DTP) tools. These macros allow the mark up of documents and the automatic extraction of requirements. The users can create, modify, and delete RDD elements, attributes, and relationships from within a commercial DTP tool. From the RDD database information is extracted using the Report Writer Facility and put in an ASCII file to be used by the text editor. From an ASCII file information is extracted by the Export Changes Facility and put into the RDD database. There are also two-way bridges between RDD and CASE tools. This allows updating the SDD with information generated by CASE tools and loading the CASE tools' database with information from SDD [92].

The RDD functions are available through a set of integrated products that may be used separately or jointly. The unified product is called *System Designer*. It encompasses the *System Analyst* tool used to model processes, the *Requirements Manager* tool featuring text editing facilities, configuration management, and report execution, the *Requirements Editor* that offers electronic extraction facilities and text editing from external source documents, and the *System Browser* tool. Information may be graphically displayed as Functional Block Diagrams, N-squared charts (matrices), and Hierarchy views. An interesting feature of RDD is

its ability to simulate process models. It allows, for example, the construction of behavioural scenarios and model interactions. This gives additional power and is a useful resource for analyzing existing (*as-is*) models and for proposing new (*should-be*) models.

8.3.2 RTM

In RTM the classes of objects are defined by the user as well as the logical associations (relations) between them. In version 3.0 the data types for attributes values are restricted to text, numbers, dates, enumerated lists, and actions [96]. The action type for attributes allows the association of a method for that attribute. In this way an attribute may be related to a file and every time the attribute value has to be accessed the associated method may place the file in an editor ready to be changed. Although external files may be associated to objects via an action-valued attribute, it is not possible to trace objects based on the contents of external files unless the external file's contents are registered in the RTM database. Also only one action may be associated to each attribute and an attribute having a value whose data type is different from action cannot have any action associated to it.

An interesting feature of RTM is the association of rules to relations. These rules can be used to characterize the way objects may be related and to control the flow of information through the development process. For example, specifying a rule to say that the cardinality of a relation between requirement and test objects is 1:M avoids the linkage of a test procedure to more than one requirement. Also, transfer rules may be specified to control the transference of links in case of expansion of objects. For example, every time a requirement object is expanded into two or more requirements the links from the original requirement to other objects may not be automatically transferred to the new requirements. Other kinds of rules may be specified. For example, a relationship 'proven by' may be associated to a rule that allows only requirements with values of 'approved' to be tested (to be linked to a test object). There are also default rules to control the existence of links if any of the linked objects are changed.

The capture of requirements from documents may be via a desktop publishing tool (DTP) in which the document is written [83]. This is possible by associating a style in the document to a class. Then every part of the document written in the specified style will be automatically used to create objects of the corresponding class. RTM version 3.0 also has macros defined for some DTP tools that allows the extraction of the information from the project database. Thus, not only the creation of objects may be done within a DTP tool but also their modification. There are also interfaces to CASE tools [84].

The extraction of information for traceability purposes is made through search mecha-

nisms to query the RTM database. The attributes of objects may be used to guide the search or to restrict the types of objects to be extracted. Also query scripts may be written and stored for later use [144]. Traceability matrices are used to show relationships among objects.

8.3.3 DOORS

In DOORS requirements are also defined as objects [115]. Their attributes are user definable as well as the relations that may exist between different classes of objects. The data type values for attributes are texts, numbers, dates, and enumerated lists. A set of objects may have global attributes. This provides ways of controlling, for example, who is in charge of a set of requirements, when was the set (any element in it) last changed, etc.

The relations may have attributes that may be used when tracing requirements [113, 114]. Traceability is achieved by queries to the DOORS database which may be filtered by the contents of the attributes. The extraction of information also makes intensive use of attributes, and sorting functions to delimit the objects of interest to be retrieved. DOORS supports a graphical representation of data which may also be used for editing, data manipulation, and querying functions.

Communication with DTP tools allows the creation and modification of requirements via those tools. There is also an export/import facility to produce/capture ASCII files to be used within other tools. Documents imported into DOORS may be marked up to delimit requirements and allow automatic extraction of requirements information. Communication with CASE tools is also possible via DOORS Application Program Interface (API).

Graphical information can be imported into documents, and printed, as encapsulated postscript or RTF (rich text format) graphics. Also, using an RTF parser, RTF graphics may be broken down into individual cells and, therefore, links may be registered to parts of a graphic or picture. The graphic may be rebuilt and exported for use by other tools or for printing.

DOORS has an extension language (DXL) that allows the user to access all DOORS functions. This may be used to automate routine tasks, perform calculations, and even to create automatic links. For example, some procedures may be defined to access data stored in objects' attributes, e.g., to calculate performance metrics on data associated with performance requirements. A number of checks are already provided, including the identification of objects with no links, with no outward links, or with no incoming links.

8.3.4 Discussion

TOOR has a number of distinguishing features with respect to the current industrial tools:

a requirements tracing language, the use of general abstract data types to specify the information to be traced, the ability to refer to information in different media in tracing queries, the use of modules for project development and to establish context for tracing, and the use of axioms for defining relationships. Below, we discuss each one of these features as well as some that are presently not supported by TOOR.

8.3.4.1 Requirements Tracing Language

The tracing mechanisms of existing industrial tools are based on queries directed to search the database used for storing information. They are in the form of proprietary search procedures, or general query languages such as SQL (Structured Query Languages) or QBE (Query By Example). Therefore they are dependent on the database structure. On the other hand, the regular expression language of TOOR depends only on the (data type) specification of objects and relations and, for example, it may be used to search the contents of external files. This is only possible using conventional database query languages if the content of the file is stored in the database itself. Also, tracing with regular expressions provides more flexibility to express patterns of related objects. Not only matches based on the objects' state and chains of relations but also sums, differences, and iterations of patterns may be expressed.

8.3.4.2 The Use of Abstract Data Types

Most industrial tools have the types of data that may be stored in their database, and therefore traced, restricted to a limited number. For example, DOORS and RTM only allow text, date, number, and enumerated list for the value of their attributes.

In contrast, TOOR allows the specification of abstract data types for the value of objects' attributes. Thus, for instance, a data type for files may be defined such that a file associated to an attribute is not just a reference pointer; it is an actual value for that attribute that may be accessed and modified via the operations specified for the corresponding data type. Moreover, since the operations specified for a data type can be used in regular expressions, external information can be actively used when tracing objects.

RTM offers a limited kind of this functionality by means of actions associated to attributes. These actions may be used, for example, to start a text editor ready to modify a certain file whenever the attribute has to be accessed. However, the result of these actions cannot be used as trace information, or incorporated in queries to the database.

8.3.4.3 Active Use of Hypermedia Information for Tracing

The definition of abstract data types and the use of operations and methods associated to them as part of regular expression queries promote the active use of hypermedia information for tracing purposes. For example, a data type `Video` for files containing video scenes may be defined together with an operation `is-there-scene` that given two `Video` files `A` and `B` returns `true` if the scenes in `A` are contained in `B` and `false` otherwise. Thus, if requirements objects are defined to have an attribute `video-evd` of sort `Video` the user may associate a video file as a value for this attribute. Moreover, if a certain requirement `req-A` is being analyzed the user may want to retrieve all other requirements possessing similar video evidence, i.e., all requirements for which the video file used as a value for `video-evd` contains the scenes of the video file associated to `req-A`. This would simply be expressed as

```
[[Requirement x if is-there-scene(video-evd(req-A),video-evd(x)) == true]]
```

Note that the retrieved requirements may not be related in the sense of having a physical link between them. The relation between `req-A` and the retrieved requirements is that they share a common piece of video evidence; in this example this relation was thought of, and expressed as a regular expression query, at the very moment of analyzing `req-A`. Of course, the above query may be enlarged to consider other kinds of chains to, for instance, retrieve all people related to `req-A` or even people related to each matched requirement.

```
[People] ToorRel [[Requirement x if
    is-there-scene(video-evd(req-A),video-evd(x)) == true]]
```

This sort of direct use of hypermedia information to trace objects is not possible with current industrial tools. Most of them have mechanisms to import/export graphics into the documents they produce but they do not use this type of information to actively trace objects. DOORS has an option to break RTF files into its constituent cells and store them for trace.

8.3.4.4 The Use of Modules for Tracing

The use of modules at a project level facilitates the definition of development information and promotes a natural scoping of activities when producing software systems. Because the application of modules to systems and projects is still under active investigation, none of the industrial tools have such concepts incorporated into their traceability procedures. Project modules are certainly less understood than specification and program modules. Much work on methodology of use, semantic definition, integration with other types of modules, etc., has to be done. Nevertheless, we believe that using modules at a project level to delimit development

phases, group actual objects, organize alternative paths in the development, define vertical structures, etc., brings to the project as a whole the same benefits as specification and programming modules brought to the specifying and coding activities. The extended example of Chapter 3 and the introductory example of Chapter 5 show how modules may be used in TOOR. In particular we have shown how modules at a project level are integrated with the other tracing mechanisms of TOOR and how they may be used to contextualize the tracing of objects.

Some industrial tools offer facilities to group objects, e.g. DOORS, and RDD is based on model definitions for the various phases of system development. However, the grouping of objects is usually made to allow relations between sets of requirements and not to delimit the scope of traceability queries. Also, the development models of RDD cannot be used as modules to provide working contexts where only the objects registered in a module, or in imported modules, may be used. The notion of traceability scope in industrial tools is usually implemented by defining what objects should be considered based on the content of their attributes.

8.3.4.5 Relationships by Axioms

TOOR actively uses the relation axioms to verify whether two objects are related or not. Relations between objects are determined not only by the existence of physical links, but also by evaluating, through the FOOPS rewriting system, the axioms used to specify them. This provides greater flexibility to define indirect links, i.e., those determined as a function of other previously defined links. This facilitates the adoption of some traceability models. For example, the role relations of Contribution Structures, being derived from the capacity of each agent with respect to the production of requirements artifacts, do not need to be explicitly made by the user (see Chapter 6). Contribution Structures may be implemented using existing industrial tools [66] since most of them have extension languages (dxl in DOORS) or programming macros (RTM) that may be used to automatically generate links based on the existing ones. However, this cannot be considered as general as the use of axioms in TOOR. Take, for example, the automatic generation of a link (a, c) as a consequence of the existence of links (a, b) and (b, c). If the link (a, b) is removed there should be a manual removal of link (a, c) or the execution of a procedure to automatically remove inconsistent links. Neither option can be said to be transparent to the user.

8.3.4.6 Support for Development

Traceability is an essential aspect of requirements management. However, it is not an end in itself and it should be part of a wider project environment which takes into account other management and technical activities like process management, including life cycle, costs, activity scheduling, personnel allocation, and coordination of multiple development teams; production of structured documents; execution of tests and verification procedures; enforcement of development methods; etc. Thus, support for traceability should be accompanied by support for other development activities.

Despite being weaker in terms of traceability features, current industrial tools offer a more comprehensive approach with respect to supporting a wider range of activities. This just points out the need for further work to extend TOOR's actual functions. Some improvements are quite straightforward while others require more intensive research, e.g., support for groupware and distributed development using TOOR's project modules. It is illustrative to comment on some of these features offered by current industrial tools.

Models in RDD are used to describe development processes and structures, the flowdown of information, and to establish the basic traceability links between development artifacts (model elements). Moreover, the same models may be used to simulate the development process itself. RDD provides facilities to create scenarios and perform simulation of process behaviour. This is a useful feature that greatly enhances the use of the tool in real development efforts. TOOR's module structure may be used to establish project and process structure with the corresponding definition of information flowdown but, as discussed in Section 3.8, this should be object of further research.

Industrial tools commonly have, in one way or another, support for groupware and distributed development. This is certainly necessary for large projects, sometimes distributed across several sites and platforms. Also, systems usually have their software part developed using established software development methods. Consequently, traceability tools have, in varying degrees, communication bridges to CASE tools where the chosen development methods can be carried out in a more specific way. Usually the communication is via import/export functions to exchange data with the tool's database. The need to communicate with other tools performing specific tasks is usually accepted as a general need and, therefore, most industrial tools have an Application Program Interface (API) enabling external tools to access their data.

Also, industrial tools pay special attention to requirements capture offering automatic parsing of requirements from external documents and integration with desktop publishing tools. the production of standard compliant documentation is a common characteristic.

	RDD	RTM	DOORS	TOOR
Automatic parsing of requirements	●	●	●	○
Baseline/Version control	⊙	●	●	⊙
Access control	●	●	●	○
WYSIWYG previewing of finished output	●	●	●	○
Groupware	●	●	●	○
Interfaces to other tools	●	●	●	⊙
External Applications Program Interface available	●	●	●	⊙
Support Open database system (standard query access)	○	●	⊙	○
Exchange of information between same-tool different installations	●	●	⊙	○
Consistency/comparison checking between same-tool datasets	●	●	⊙	○

Table 8.5: INCOSE Survey of Requirements Management Tools. The answers are given by the vendors according to the following legend: ● full; ⊙ partially; ○ no compliance. The last column is not part of the original survey.

These points are illustrated in Table 8.5. The table reproduces some answers given by the vendors of each tool to a recent survey promoted by the International Council on Systems Engineering (INCOSE) [74]. The points in the table clearly indicate the need for comprehensive requirements management solutions where traceability, although fundamental, is just one aspect of a more general management need. The current state of the practice somehow indicates the need for integrated solutions, which benefit from development coming from several specific areas. It is likely that the whole process will be largely influenced by improvements in document management systems, CASE tools, and integrated project environments (IPSE).

Document Management Tools

The production and control of documentation is a critical aspect not only because documents are widely used as a communication means but also because milestones in system management usually have a physical counterpart in terms of documents, e.g., indicating the completion of some phase. Many requirements management tools already provide some support for documentation. For example, they usually are centered on, or have facilities for, the production of structured (standard compliant) requirement specification documents. They also usually have facilities to mark up documents for requirements extraction. However, they do not provide support for all activities related to the production and use of documents in general. This is recognized in the existence of interfaces to DTP tools from which requirements can be created and edited. The functionalities provided by the most popular DTP tools are unparalleled by the native text editing capabilities of existing requirements management

systems. Many text editing tools are even used as stand alone tools for the production and management of requirements documents. This is certainly due to their ease of use but there are also some aspects that, at present, are only considered in more general kind of document management systems. For example, the definition and control of the flowdown of documents (distribution) across an organization's structure, the control of documents status, whether in production, under elaboration, approved, or in the n*th* revision, the assignment of personnel responsibilities, etc. Document management systems may provide a useful support to automate these aspects and to integrate them with other activities of system development.

CASE Tools and Integrated Project Environments

The automation of software activities is essential for the development of systems of a certain size/complexity. The methods chosen for the various phases of system development require discipline, consistency, and integrity regarding to their steps and information. This, for large amounts of data, can easily become unmanageable if done manually. CASE tools are used to assist the accomplishment of these tasks for the method they are designed to support. Requirements and design are also supported by (upper) CASE tools. Traceability is achieved as a by-product of their use, since the several kinds of information are integrated in one way or another and they usually offer various ways of extracting information. Nevertheless, traceability is not their primary concern and, therefore, better support is still given by specific traceability tools. This is reflected in the existence of interfaces to CASE tools from almost all requirements management systems. The integration is usually made by means of import/export functions used to transfer data from one environment to another.

No matter how efficiently CASE tools may be used to support (a set of) development activities, it is argued that the current generation of such tools are not adequate to support the *way in which the techniques are applied* [95]. This, together with a need for context sensitivity when using any development model, contributed to the development of meta-CASE tools [35]. They are designed so that they may be customized for each application. Thus, their use is tailored to fit particular circumstances. The tasks, elements, and control may be defined to generate in fact a different CASE tool for each occasion of use. Traceability is enhanced as the information available to trace and its production is more pertinent to what is expected by the people using these tools.

The trend toward activity support leads naturally to the use of process-oriented models of system development, also called IPSE (Integrated Project Support Environment) [68]. These models are centered on the definition of the relevant processes and their interconnections. Also, they are used to enact the actual processes and, therefore, guide the development

and co-ordinate the use of the necessary tools. This provides a more disciplined, therefore manageable, approach to the execution of development activities.

This form of automated support provides for the capture of trace information related directly to the way information is produced and used. This contrast with more static information available in data-centered models of development. Research on IPSE has been going on for a long time but it is still very strong and active [18, 46]. Among several specific topics of research (see [141] for a survey of IPSE systems and [46] for a more detailed presentation) one important aspect of is that of tool integration [17]. This is of direct concern to most of the current requirements management systems since they all rely on third party tools to automate one aspect or another of the development [92].

Our traceability model is in accordance with the trends towards more comprehensive and configurable software engineering environments. Because TOOR can specify traceability structures which are realized differently depending on the data types, methods, and axioms defined for each class in a project specification, it provides a solid foundation for a traceability system that can work with information produced by different tools. This integration is only partially achieved in the current version of TOOR, and further work will be needed.

8.3.4.7 Summary of Similarities and Differences

The main differences between TOOR and industrial tools reside in TOOR's abstract features. Indeed, TOOR's traceability model may be viewed as consisting of three languages: to specify the structure and type of objects and relations, to express patterns of related objects, and to define traceability scope through specification of project modules. As discussed in the previous paragraphs and exemplified throughout this thesis, the characteristics of each language such as the use of axioms, abstract data types, direct reference to external files, regular expressions, etc., and their interconnection accounts for much of the flexibility and expressive power of TOOR.

As observed in the preceding sections, the traceability support offered by industrial tools is not as general as in TOOR. They trace what is available in its internal structures according to the formats prescribed for the stored information. In contrast TOOR traces what is specified to be traced. The actual tracing in TOOR is also performed on available data but the fact that the objects and relations are specified by means of a formal specification with executable capabilities, makes TOOR's approach more general in principle. The difference resembles that between a programming language and the programs built using it. In TOOR one can actually 'program' the tracing procedures and the tracing environment itself, including the overall structure of objects and relations. Defining the tracing structure using abstract data types

with their associated (executable) operations governed by axioms achieves flexibility greater than simply declaring the object structure by means of user-definable attributes. Even the user-definable models of RDD are not as general because they lack the abstract data type capability to specify the elements comprising a given model.

The trace extraction features of TOOR are also of a general nature. They are centered around a regular expression based language used to specify patterns of related objects. The pattern matching procedure of TOOR's traceability language goes beyond the simple identification of identical symbols (objects). It is extended by several regular definitions and, through the use of FOOPS axioms, it takes into account the actual specification of objects and relations. This in a certain sense can be viewed as more general than the database query languages used in industrial tools. First, by using the specification of objects and relations in regular definitions, TOOR's regular expression language is made abstract in that its particular pattern matching procedures are defined according to specific instantiations of the model, i.e., according to particular project specification used. Second, for the same reason (axioms and abstract data types), the tracing language in TOOR can search information stored outside the database.

Note that the existence of application program interfaces and macros and definition languages in industrial tools does not provide the same flexibility in terms of model definition. They basically allow the programming of procedures acting on the defined structures. On the other hand, they make possible the specification of checking and verification procedures which is not presently implemented in TOOR. This observation points us to another line of comparison concerning the production of traces and the overall support provided for system development. At the moment, TOOR does not provide the same kind of development support present in commercial tools, e.g., distributed development, communication with CASE tools, extraction of requirements from textual documents, etc. However, these points do not impair our traceability model. As explained before apart from the need to support distributed development, which should be the object of further research, the other points, although important to practical development, may be viewed as auxiliary insofar as they do not impinge on the traceability model.

One source of confusion we face when presenting TOOR's features is the distinction between the tool and the model. Our traceability model is abstract in nature: it should be instantiated by an specific project specification to be made operational. TOOR just implements it, naturally reflecting the strengths and weakness of any given instantiation, i.e., any concrete traceability model that is specified. Of course, as a tool TOOR constrains the model and also its instantiations. The limitations in implementing specific traceability models in

TOOR are discussed in Chapter 7. The distinction between the tool and the model and how one affects the other is more fully explained below in our evaluation section.

Chapter 9

Conclusions

We conclude the presentation of our traceability model by discussing some possible future work, by arguing in favor of the satisfaction of the requirements we set up in Chapter1, and by presenting a critical evaluation of its features.

9.1 Future Work

TOOR implements a general traceability model based on the transformation of a given configuration of objects and relations into an automaton used to match patterns described by regular expressions. The traceability model also addresses the need for user-definable features, the need to account for the structure of development, the need to use information stored in different formats, and the need for varied and flexible modes of tracing. Several extensions may be sought to improve existing features and to add new ones:

- TOOR embodies a single user model of traceability. Although this is sufficient for projects with a unified database and a centralized update of objects and relations, it is certainly not adequate for distributed or concurrent developments where groups of people may work on different aspects of the same configuration. A first extension would be to make TOOR a multi-user environment. Additionally, the traceability model itself may be extended to support multiple viewpoints [99, 44]. This would require the coordination of different configuration states and mechanisms to trace objects according to specific viewpoints. This approach should also address the related aspect of managing (tracing) inconsistencies that naturally occur in a multiple view perspective [41, 45].

- The module structure of TOOR makes possible the establishment of a policy for object creation and object use. However, it does not have modularization constructs other than the import mechanism. Specifically it lacks constructs for combining existing modules, for parameterization, and for using existing modules by renaming (some of) its features.

As it is, TOOR's modules provide a clear separation between modularization for the project and modularization for the system being developed. The latter is achieved through the use of FOOPS modules. Nevertheless, it would be worthwhile investigating a more uniform approach with common module features applying to both the formal and informal aspects of software development, along the lines of the hyperprogramming paradigm [51, 53].

- The management of different configurations, even for a single user project, is a valuable aid for tracing objects. It helps if one wants to trace back to objects or versions of objects defined in different baselines or if the project involves the development of different versions of the same objects, e.g, for implementation in different machines. The configuration management facilities of TOOR are limited to the use of `Abandon` and `Replace` relations to maintain old objects available for tracing, to the use of (nested) modules to define boundaries for alternative developments, and to the use of object's modification dates to trace modified objects. The configuration management of TOOR could be extended to provide specific features for the definition and restoration of project baselines. Alternatively, the traceability model itself could be modified to use hypergraphs instead of graphs to represent configuration states. This would provide support for version control along the lines presented in [88].

- A critical aspect for an efficient use of the traceability model of TOOR is the definition of an adequate project specification: the careful specification of the classes for objects and relations that may be used in a project. There are several proposals defining specific sets of relations among software related objects and defining domain models containing specific set of relations. This seems clearly to be dependent on several factors, including the characteristics of the project and the situation in which it is carried out. An interesting direction of research is to use the formal user-definable features of TOOR to investigate useful sets of relations and their applicable contexts.

- It also would be interesting to investigate the application of TOOR's traceability mechanism to domains other than software development. Two domains of application would be theorem proving, to keep track of development of large proofs, and document preparation, to keep track of document changes. To this end it may be necessary to extend TOOR in order to associate commands to trace results, in the line of the *command* declaration of the MAKE system.

9.2 Revisiting TOOR's Requirements

Requirement *The tracing model should not be tied to any particular way of developing software.*

> The module structure of TOOR supports the assignment of classes of objects to specific modules. It also supports the specification of a policy for creating and using objects inside modules. Therefore, it is possible to implement a module structure adequate to various life-cycle models with modules resembling life-cycle phases and 'separation of concerns' ensured by the specification of the classes whose objects may be created or only used in a certain module/phase. On the other hand, although our tracing model is based on general concepts its implementation is directed to the use of FOOPS as a language to specify and develop object-oriented system.

Requirement *It should be possible to associate trace units to phases of the development process.*

> The phases of a life-cycle model can be operationalized in TOOR by assigning to each phase a specific module and defining the module to allow the creation and use of objects concerning its assigned phase. Traces in TOOR are viewed as relations among objects and relations themselves are implemented as objects of particular classes (subclasses of `ToorRel`). Therefore, both relation classes and object classes may be defined to have their objects created only within the context of a specific module/phase.

Requirement *It should be possible to define a forward ordering on the phases of the development process. It also should be possible to define traces of trace units associated with one development phase on trace units associated with the subsequent phases.*

> A forward ordering on the phases of a development process may be indirectly specified through the assignment of classes to modules. The definition of a class `Requirement` to have its objects created only in a module `REQ-ELICIT` and the definition of classes `Allocated-to` and `DesignUnit` to have its objects created only in a module `DESIGN` which also can use objects of `REQ-ELICIT` reflects a natural forward ordering from module `REQ-ELICIT` to module `DESIGN`. However, there is no mechanism implemented in TOOR to *impose* such an ordering. The definition of traces of a given object on another, where the objects belong to different phases, is illustrated by the example above: the `Allocated-to` is a

relation that captures the traces left by `Requirement` objects on `DesignUnit` objects.

Requirement *It should be possible to follow traces according to the backward ordering of phases.*

Defining a forwarding ordering on phases/modules implies the definition of a reverse backward ordering. Therefore, the specification of traces and trace units with respect to the forward ordering also applies, in a reverse way, to the backward ordering.

Requirement *It should be possible to define traces that occur in the context of a phase, i.e, traces involving only trace units associated with the same development phase.*

This is achieved by the definition of relation classes whose objects have their source and target objects from the same module.

Requirement *It should be possible to define traces involving trace units not associated with any phase of the development process.*

Even if a project is structured with modules corresponding to development phases it is possible to define some classes whose objects may be created in the context of another module not related to any phase. This module can be imported by several others in such a way that its objects may be used in several phases of the development.

Requirement *It should be possible to group trace units that are associated with different development phases without redefining the development phases. It also should be possible to define traces involving trace units grouped in this way.*

The argument for the previous requirement also may be used to support the definition of trace units that are associated with different development phases: a class may be defined in a module not related to any development phase and be used in any other module.

Requirement *It should be possible to redefine the development phases and the ordering on them. It also should be possible to redefine the associations between trace units and phases or to define multiple association among them.*

The definition of modules in TOOR is not fixed at the beginning of a project. The module structure of a project may be modified dynamically; new modules may be

created and existing ones may be removed as necessary. Also, module signatures can be modified.

Requirement *It should be possible to adapt the model to particular project environments.*

A project specification requires careful thought to properly establish the necessary objects and relations that are useful for a given way of developing systems. Also, given the nature of what is specified — the objects used to develop systems and *not* the objects related to the domain of the system being developed — the project specification is relatively stable and is intended to be used for families of projects. Indeed, a company may have one single project specification for all of its projects: it may contain an `Interview` class if the company's analysts are used to interviewing people and it may not contain a `VideoScene` class if video analysis is not part of the company's culture. Nevertheless, the TOOR tracing model is adaptable to particular situations. Different project specifications may be defined for different companies and even for different projects in the same company. It is also possible to modify a project specification for a particular project even if the project is in the middle of its development.

Requirement *It should be possible to retrieve trace information in different ways.*

The trace extraction facilities consist of three trace modes: tracing through module structure, tracing by browsing, and tracing through regular expressions. Each one of these modes has several alternatives: there are different ways to browse objects as well as different ways to trace using module structure and regular expressions. The project specification gives additional flexibility through the use of axioms to specify relations. By changing axioms one can change the indirect links between objects and, therefore, the ways objects are related. This can be used to trace objects according to relations not previously defined. For example, one can produce new information by simply adding an axiom to specify that an existing relation should also be viewed as a composition of two others.

Requirement *It should be possible to define traces and trace units representing things that lie outside the technical chore of software development.*

Classes in a project specification may be defined to represent any concept. Therefore, any object may be modelled and traced.

Requirement *It should be possible to register as trace units the actual artifacts used in trace production.*

> The project specification allows the use of external files as attribute values. The sort `T$File` provided in the standard project specification is used by TOOR to make the necessary association. In this way, a `Program` object may have its content attribute directly related to the file containing the actual code and a `Design` object may be directly related to the actual design chart.

The above requirements are of a general nature and are not precise enough to be tested. Nevertheless, the rationale given in Chapter 1 is sufficient for a reasonable understanding of the main goals they are intended to address: *flexibility* and *adaptability*. Our arguments in favour of the satisfaction of these requirements by TOOR and the points highlighted in the presentation of this thesis, also reflect our primary concern to develop a model with *flexible* traceability features and that is *adaptable* to a variety of development situations.

9.3 Evaluation

Our traceability model may be understood as comprising a graph structure of related objects, a language to express relationships, and a modular structure providing scope to both the creation and inspection of objects and relations. The graph structure is defined by means of a formal specification and therefore it is not established beforehand: the precise structure as well as the constraints imposed on the objects and their relationships are defined by a project specification and may vary from project to project. Also, the module structure varies from project to project. As explained in Chapter 3, although transparent to the user, modules are created and composed according to a precise syntax. Indeed, a second way of viewing our model is to look at it as consisting of three languages: a language to define structures of related objects, a language to express relationships, and a language to define module structures. Therefore, our model is abstract in nature and a practical evaluation will depend on particular instantiations. Using our model to implement an IBIS-like structure of related objects is different from using it to implement a structure of related objects following the Contribution Structures approach.

TOOR implements our traceability model. It realizes the model and at the same time constrains it. The release discussed in this thesis[1] is a prototype implementation but it operationalizes much of the benefits of using our model for traceability purposes. Therefore,

[1] TOOR is currently in version 0 (version 1 is intended for when it is somewhat more stable than its current prototype implementation). The release discussed in this thesis is release 2, being the third of a series that began in 0 each one intended to implement a particular aspect: version 0.0 implemented the communication with FOOPS and the generation of templates, version 0.1 implemented the hypermedia features, and version 0.2 implements the module structure, including the interactive tracing mode, and the regular expression based mode. This last version incorporates some of the features implemented in the previous one. It is the version intended to evolve to version 1 and is available via ftp: ftp.comlab.ox.ac.uk in directory /pub/Packages/TOOR.

TOOR may be used to evaluate the *use* of the model. For evaluation purposes the distinction between abstract features of a model and their realization is not always clear and may even not be desirable. Some evaluation points are affected in a overlapping way by both the abstract features and the constraints imposed by an actual implementation. For example, usability is normally considered with respect to (acceptable) implementations but it may also be considered against intrinsic features of a model, e.g., the requirement to use a fixed set of relations between objects that is not natural in terms of users' perception. In the latter case, a particular implementation may be shown to minimize some negative model feature, e.g., by automating relations that are not intuitive. Accordingly, in the discussion below we refer to TOOR meaning both the tool and the model it implements. Most of the time this duality is appropriate. For particular points where we want to make the distinction clear and the context is not enough, the words *tool* and *model* are used.

9.3.1 Validity

In this thesis we presented our model as it is: an abstract traceability model. Its abstract features are discussed in full through the various languages comprising it.

In Chapter 2 we give the details of the project specification, showing how a FOOPS specification can be used to set up the tracing environment. The necessary constraints and each language feature are explained and examples of their use are given. The semantics of the specification language used is outside the scope of this thesis but we comment on some semantic issues like parameterization and the underlying term rewrite system, and we give references to its definition elsewhere.

In Chapter 3 we discuss the syntactic features of the language used to create project modules and establish tracing boundaries. Again, each feature is explained with localized examples showing how they work. The formal descriptions necessary for a correct manipulation of module signatures are given in Appendix A. The incorporation of project modules in system development is discussed through an extended example.

In Chapter 4 we present our regular expression language. Its internal mechanisms are extensively described through the formal definitions of the grammar and automata. We use an extended example to illustrate how regular expressions work with a set of related objects and their formal specification. Appendix A gives details of particular extensions described in the text.

This kind of presentation is normally taken when presenting language features. We believe that it is appropriate for showing the language capabilities. Moreover, through the formal definitions one can verify whether the claimed features may be accomplished or not. For

example, in our case it is clear that the FOOPS rewrite system can be used to evaluate axioms at run time; that the communication between TOOR and FOOPS, as described in Chapter 2, can assess what is specified in a project specification; that from a set of objects and relations automata can be built such that they recognize sequences of related objects as described in Chapter 4, that the FOOPS axioms can be used as part of the matching procedure, etc. However, our model is abstract but it is not of a general purpose; it aims to be used for system development, instantiated by actual traceability models. Therefore, it remains to be shown that the instantiation process can result in a effective environment for tracing. To this end we present in Chapter 5 the abstract data type features used to access hypermedia information and in Chapter 6 we discuss how relations can be defined to keep track of project evolution. The instantiation process itself, although illustrated throughout the thesis with small examples, is given a more focussed attention in Chapter 7. There we illustrate the use of TOOR to instantiate the IBIS, REMAP and Contribution Structure models. This is done in a fragmentary way but enough to show the working features of TOOR.

The demonstrated features of our model show that it meets our claim for flexible structuring facilities and for tracing information not thought of in advance. Also, given that TOOR may be instantiated with interesting, and accepted traceability models, we argue that it can be used for real software development. This goes beyond the simple effectiveness to define and keep track of related objects. We should comment on issues like organizational impact, usability, scalability, etc. These are the topics of the following sections.

9.3.2 Requisites

We begin discussing the requisites necessary for an effective use of TOOR. First, the definition of a project specification should certainly be attributed to a person skilled in the use of formal methods, and FOOPS in particular. This difficulty is ameliorated by using the standard project specification as discussed in Chapter 2. Also, in real settings the definition of a project specification is likely to be attributed to a single person or group in the organization. This is in conformance with similar practices in the database community, where a database administrator (group) is usually responsible for the administration of database issues. Also, the existence of a single person or group responsible for producing project specifications is beneficial to keep the structure of the tracing information attached to the organizational practices. For example, trace information that interests the upper management levels can be specified for a whole range of projects in a uniform way. In this respect it is interesting to note that being bound to organizational practices in terms of the objects and relations specified for a project does not preclude the use of the tool in a situated way. Our model

does not limit the ways a tracing query should be expressed and it is certainly possible to fix some parts of the specification for a project while leaving others open for localized use. For example, a general module structure may be defined for a certain organization consisting of `Elicitation`, `Specification`, `Design`, `Implementation`, and `Test`. Each of these modules will have their signatures fixed so that the objects and relations that may be registered are pre-defined. This may be good for the organization as a whole since it promotes uniformity. However, nothing prohibits the definition and creation of new modules for particular projects. For example, if video-based elicitation is not part of the organization culture but a particular team wants to try it for a certain project then a `VideoElict` module may be created inside (or imported into) the previously defined `Elicitation` module.

Second, our traceability model, being generic, needs to be instantiated with a particular model of related objects. This is accomplished by providing a suitable project specification defining what are the objects and relations to be used, and also what are their constraints. Chapter 7 provides an extensive discussion of this point. As we stress in the sections below the choice of a particular way to model related objects affects the use of TOOR in different ways. For some aspects like usability and organizational impact using TOOR to implement Contribution Structures is different from using it to implement REMAP or any other IBIS-based model.

9.3.3 Usability

The usability of the tool should in fact be assessed by its use in real settings and, therefore, we cannot ascertain definitive points. However, there are some points that we can be reasonably sure of. For example, a breakdown analysis [127] to assess disruptions in the way users perform tasks using the tool should, to some extent, reveal the same strengths and weakness of existing requirements environments. The use of TOOR in terms of user operations does not differ significantly from the operation of current traceability tools: the objects are entered and edited by means of templates and the several views are accessed by intuitive graphical interfaces.

Of course, as discussed in Section 8.3.4, some standard features like the automatic extraction of requirements from external documents and the communication with other well established tools are missing in TOOR. This certainly affects usability (for an interesting example of user preference to produce initial, tentative, requirements documents using a well known text editor instead of a given requirements management tool see [42, Chapter 5]). However, these are missing implementation features and should not be regarded as limitations of our traceability model since they may be implemented without changing its basic

features. On the other hand some inherent features of our model like the automatic generation of links based on the axiomatic definition of relations and the ability to define general data types for the content of object's attributes should enhance usability, for they allow the definition and production of sophisticated trace information requiring no more effort than that necessary to register objects and simple links.

The production of traces is dependent on their types. For the most usual, artifact-based traces, the work involved is straightforward consisting of filling in the respective templates. Some may even be automatically generated and the facilities provided by TOOR helps the specification and the actual generation of such traces. For more complex traces, e.g., those involving personnel based relations, the production is also of a more complicated nature. They rely on the collaboration of all people involved and on the adopted organizational practices. Despite the inherent difficulties, one such model (Contribution Structures) is evaluated as being possible of effective implementation [66] (maybe coupled with changes in management and development practices) using current requirements management tools. Therefore, we argue that TOOR may be used just as well and, given the reasons discussed in Chapter 7 and Section 8.3.4, even better.

9.3.4 Organizational Impact

The organizational impact of adopting our traceability model is also dependent on particular characteristics of the environment in which it may be used [122]. Analysis like the *user cost-benefit assessment* method [39], the use of scenarios to explore implications, etc., are meaningful only for the specific contexts where they are carried out. However, we may again make some general considerations. It is usual to identify the potential 'winners' and 'losers' of a new technology or method. In our case, it is clear that the upper management are among the winners. They are usually involved with the extraction and interpretative aspects of traces and few demands are put on them with respect to trace definition and production. The navigation features of TOOR through the use of modules and browsing windows are simple. For more complex queries, a regular expression may be defined (maybe with external help) and stored for repeated use. It should be observed that those fixed (stored) queries may be in fact of a very general nature, e.g., *all requirements that are late in their allocation date* or *requirements that have been reopened for discussion*, the result of each actual tracing varying according to the status of the project, i.e., the actual registered objects and relations. Also, the use of modules for tracing is a useful management tool. It offers the possibility to leave out (modules used for) the more technical aspects of traces, like programming related objects, and to focus attention on (modules used for) aspects like personnel and test related

objects.

Requirements engineers, analysts, and designers are in an intermediate situation. They gain because the actual artifacts used by them in performing their tasks may be directly used as objects. Also, since the modifications they make in the artifacts are automatically available for trace, by accessing the contents of external files via operations defined for file data types, they can have more confidence in the timeliness and accuracy of the information. The drawback is that the workload will increase since not only the objects should be registered but also their relations to other objects. The transference of this work to documentators can be made but care should be taken because the use of the right modules necessary for an appropriate registration of traces may not be meaningful to people not having a clear picture of the project structure.

TOOR provides facilities for the automatic production of some traces. It also minimizes the need to actually register all defined relations since some of them are computed using FOOPS axioms. Nevertheless, accurate registration of objects and relations is still necessary for information to be reliably traced. For large projects this work is usually left to be done by clerical and administrative staff that may not see the benefit of their additional work. Those are the people likely to perceive themselves as 'losers'.

9.3.5 Adaptability

For adaptability we refer to TOOR's ability to be used in different situations and for different types of information. This is connected to our claim that TOOR provides some support for situatedness.

Situatedness, for software development, is usually translated in terms of flexibility and adaptability with respect to the interactions between the environment, the tasks, and the agents involved. Hoffman et al. [70] state that a situated design is basically a flexible design approach leaving room for change and adaptation. Goguen [62] provides a list of *qualities of situatedness* that is also used in a more general account for a socially situated theory of information [58]. We claim that TOOR possess or promotes, to some extent, some of the qualities mentioned in [62], namely those of being *local*, *contingent*, *open*, and *vague*. Briefly, from [58], locality means that events are seen as such in a particular context, including a particular time and place; contingency means that the construction and interpretation of events depend upon the current situation, potentially including the current interpretation of prior events; openness means that accounts of events cannot in general be given a final and complete form, and vagueness means that practical information is only elaborated to the degree that is useful to do so. Our claim that TOOR promotes situatedness is based on the

following specific points.

- Locality, contingency, and vagueness. The existence of a language for tracing enables the expression of the same thing in different ways. Different perceptions may happen by virtue of different positions in the organizational structure, different commitments to the object being traced, etc. Using regular expressions, the structures (i.e., the automata) used to trace and resulting from it are dependent on the configuration state and on each query. The use of a regular expression based tracing language promotes situatedness because a need for tracing may be expressed in different ways by different people; the language has special constructs (regular definitions) allowing the specification of matching procedures with different degrees of precision; and, using the incremental regular expression search, the queries may be constructed to consider previous results.

- Locality and contingency. Modules are defined to hold objects and apply to a project level. Therefore, they may be used to delimit software life-cycle phases, organizational structure, and, generally, workplaces where the objects that may be used are registered. All participants in a project orientate themselves to the particular module structure defined for the project. This orientation is somehow enforced since objects may only be used in modules from which they are visible and may only be created in modules for which they are designed. Of course, particular people may know only particular parts of the module structure. The use of modules to establish a context for tracing promotes situatedness because people may restrict the trace to parts of the project that are meaningful for them and one tracing query may produce different results depending on the (module) context of the query. Therefore, we have the possibility of deliberately (and dynamically) enlarging or reducing the search space depending on the situation at hand. Moreover, the objects registered in modules are supposed to be the actual objects used in the development and not just a documentational reference to them.

- Openness and contingency. The use of axioms to define relations provides flexibility to change the ways objects are related to each other. This promotes adaptation to different situations and also may be used to test different perceptions of the same situation.

- Contingency. The direct use of information stored in external files and in different media allows tracing information that initially may have not been thought to be relevant. This is a special case of the use of abstract data types to specify the value of object's attributes. This promotes situatedness because information may be traced as soon as it is available. If an external file is edited its contents may be used in tracing queries with no need to register, or transfer, the modifications to TOOR's database.

9.3.6 Effectiveness and Scalability

The use of our traceability model to effectively register and trace information may be assessed from two different points of view.

First, the formal definitions of its constituent parts, namely the FOOPS specification of objects and relations, the configuration grammar, the configuration state, the configuration automaton, the module system, and the regular expression based language, allow the verification of its claimed properties. For example, the determination of indirect links between objects as a result of specifying relations by axioms is supported by the term rewrite system of FOOPS. Also, given a certain configuration of objects and relations, its transformation into a configuration automaton and the assurance that this automaton can recognize sentences of our grammar may be verified through the definitions given in Chapter 4. The formalization of the model also allows for an assessment of its limitations. For example, since regular expressions denote regular sets it is not possible to express non-regular patterns. A classical example in compiler literature is the expression of balanced parenthesis. In traceability terms this may be paraphrased as the impossibility of expressing sentences like *all objects related to an equal number of other objects in a forward direction as well as backwards*, i.e., to retrieve relations of the form

$$b_1 \rightarrow o \rightarrow f_1$$
$$b_1 \rightarrow b_2 \rightarrow o \rightarrow f_1 \rightarrow f_2$$
$$\ldots$$

A second way of verifying the effectiveness of the model is to verify if it may be applied to real world environments. To this end we just recall that our traceability model is of a general nature and does not embody a model for the application domain. Rather, it is intended to be configured for use in different application domains. Each actual configuration is determined by a particular project specification. Therefore, the use of our traceability model to effectively trace information in real systems development is dependent on the model of the application domain defined by the project specification. The examples in Chapter 6 illustrate the possibility of using interesting, and somehow accepted, models of objects and relations. Of course, after having decided to use a certain model for the objects and relations, the question remains of how effectively the model can be realized in TOOR, i.e., how effectively the information required by the model can be registered and accessed. Again this is highly dependent on the particular model being used. As discussed with respect to usability and organizational impact, the use of TOOR (in terms of operation) is comparable with the use of current requirements tools. However, as exemplified for IBIS and KJ, in Chapter 7, not all functionality required for an effective use of every model of related objects may be implemented in TOOR.

For scalability we have two sides. On the technical side we have to assess the construction of the configuration state and the configuration automaton. For each traceability query we have to assess the construction of the regular expression automaton and the construction of the intersection automaton. The construction and maintenance of these structures is simple. For example, for the configuration automaton, creation and deletion of objects require fixed time consisting of the addition (or deletion) of a state and a transition from the initial state to the newly created one. If it is decided not to maintain the structures dynamically, but to build them every time they are needed, we also have linear time on the number of objects and relations. The space is also linear and depends on the particular structure defined for the objects and relations. The use of regular expressions for querying the configuration state is practical. It usually requires quadratic time on the size of the expression to construct the corresponding regular expression automaton, and it is shown that for deterministic regular expressions this time may be even linear. The match between a regular expression automaton and a configuration automaton may be viewed as a simple graph matching algorithm. The use of modules presents no great demand on the system and the other functions like browsing may be implemented using available techniques for traversing data structures. The other side of scalability, that is, whether TOOR scales in terms of non-computational resources or not may be addressed using arguments similar to those employed previously when discussing usability and organizational impact.

The best way of using our traceability model is for the work to be carried out through tools already necessary to perform the natural activities of system development. This points to its use in an integrated environment where the production of each type of artifact may happen through specific editors. In this respect our model is particularly suitable since its internal structures may be dynamically constructed and do not depend on any pre-defined way of storing objects, except for a clear identification of the objects and relations to be traced. The mechanisms of our traceability model, in particular the use of regular expressions to trace objects, may be made to work with different environments. However, we designed TOOR having in mind the hyperprogramming paradigm of software development [56, 53, 51], as reflected in our module policy and in our use of FOOPS objects.

Appendix A

Formal Aspects of TOOR and Deferred Proofs

A.1 TOOR Modules

TOOR modules are not specification modules. In Particular, they do not specify classes. They just specify *rights* to create and use objects. Objects in TOOR are created in a unified database but they should be registered in a module. Objects may be used either in the module they are registered in or in any other module importing the module in which they are registered. The signature of a module specifies the classes of objects that may be created or used although the actual use of a particular object depends on whether the object is visible or not. The points below formalize these concepts.

A.1.1 Module Signatures

- The signature of a module M, denoted by $\Sigma(M)$, consists of the classes of objects that may be registered or used in M. To differentiate these two aspects we partition $\Sigma(M)$ into two sets: $\Sigma_R(M)$ is the set of classes from $\Sigma(M)$ whose objects may be registered in M, and $\Sigma_{\overline{R}}(M)$ is the set of classes from $\Sigma(M)$ whose objects may not be registered in M. A class may be used in a module M if it is part of either $\Sigma_R(M)$ or $\Sigma_{\overline{R}}(M)$. Therefore, denoting by $\Sigma_U(M)$ the set of classes that may be used in M, we have:

$$\begin{aligned} \Sigma(M) = \Sigma_U(M) &= \Sigma_R(M) \cup \Sigma_{\overline{R}}(M) \\ \Sigma_R(M) &\cap \Sigma_{\overline{R}}(M) = \emptyset \end{aligned}$$

- A class is said to be declared (imported) for object creation if its objects may be registered in the module declaring (importing) it. Similarly, a class is said to be declared (imported) for object use if its objects may be used, but not registered, in the module declaring (importing) it.

- All classes declared in a module by means of a **tclass** declaration are considered declared for object creation. That is, in a module M, we have

$$\{c \mid c \text{ is a class declared by } \texttt{tclass}\} \subseteq \Sigma_R(M).$$

- If a module N is declared inside M, denoted by $N_{\sqsubseteq M}$, we have

$$\begin{aligned} \Sigma_R(M) &\subseteq \Sigma_R(N_{\sqsubseteq M}) \\ \Sigma_{\overline{R}}(M) &\subseteq \Sigma_{\overline{R}}(N_{\sqsubseteq M}) \end{aligned}$$

- In TOOR, when a module imports another one what is really being imported, in terms of signatures, are class declarations. We use the notation $M^{N \lhd C,U \rhd}$ for a module M importing the classes in C and U from a module N. The set C consists of all classes imported from N for object creation and the set U consists of all classes imported from N for object use. We also use M^N to denote $M^{N \lhd \Sigma_R(N), \Sigma_{\overline{R}}(N) \rhd}$.

- If a module M imports a module N, $M^{N \lhd C,U \rhd}$, the imported classes of N are incorporated into the signature of M in a way such that

$$\begin{aligned} \Sigma_R(M^{N \lhd C,U \rhd}) &= \Sigma_R(M) \cup C \\ \Sigma_{\overline{R}}(M^{N \lhd C,U \rhd}) &= (\Sigma_{\overline{R}}(M) - C) \cup (U - \Sigma_R(M)) \end{aligned}$$

  - We note from the above that classes declared for object creation take precedence over classes declared for object use. The classes that could not be registered in M and that are imported for object creation are removed from the use only part of the final signature, as well as the classes that are imported for object use but that were already declared for object creation.
  - However, the order of importation does not alter the final signature. If a module M imports module P and Q, then

$$\begin{aligned} \Sigma_R((M^{P \lhd X,Y \rhd})^{Q \lhd W,Z \rhd}) &= \Sigma_R((M^{Q \lhd W,Z \rhd})^{P \lhd X,Y \rhd}) \\ \Sigma_{\overline{R}}((M^{P \lhd X,Y \rhd})^{Q \lhd W,Z \rhd}) &= \Sigma_{\overline{R}}((M^{Q \lhd W,Z \rhd})^{P \lhd X,Y \rhd}) \end{aligned}$$

  - Also, there is no need to rename imported classes. All classes in TOOR are uniquely specified by the project specification and, therefore, identical class names declared in different modules refer to the same class.

- TOOR has four different ways of importing modules. They differ in the way the imported classes are selected and preserved with respect to their original declaration.

- The first way of module importation is by means of the **import** declaration. In a module M, every declaration of the form `import N` is equivalent to $M^{N \lhd \Sigma_R(N), \Sigma_{\overline{R}}(N) \rhd}$.

- The second way of module importation is by means of the **use** declaration. In a module M, every declaration of the form `use N` is equivalent to $M^{N \lhd \emptyset, \Sigma_R(N) \cup \Sigma_{\overline{R}}(N) \rhd}$.

- The third way of module importation is by means of the **from/import** declaration. In a module M, every declaration of the form `from N import c1 c2 ...` is equivalent to $M^{N \lhd C, U \rhd}$, where

$$C = \{c \mid c \text{ is declared in } \texttt{from/import} \text{ and } c \in \Sigma_R(N)\}$$
$$U = \{c \mid c \text{ is declared in } \texttt{from/import} \text{ and } c \in \Sigma_{\overline{R}}(N)\}$$

- The fourth way of module importation is by means of the **from/use** declaration. In a module M, every declaration of the form `from N use c1 c2 ...` is equivalent to $M^{N \lhd \emptyset, U \rhd}$, where

$$U = \{c \mid c \text{ is declared in } \texttt{from/use} \text{ and } c \in (\Sigma_R(N) \cup \Sigma_{\overline{R}}(N))\}$$

- A class may be declared as private in a module. Obviously, a module imports only the classes that are not private in the imported module. The notation above may be easily extended to incorporate this concept.

A.1.2 Module Structure

The module structure of TOOR is described by a directed acyclic graph $H = \{N, E, v_0, v_1\}$ where N is the set of nodes with each node corresponding to a module, E is the set of edges, and v_0, v_1 are functions giving for each edge its initial and final nodes. The set of edges is partitioned in two sets, E_I and E_S. There is an edge $e \in E_I$ from node B to node A ($v_0(e) = B$,$v_1(e) = A$) if and only if module A imports module B. There is and edge $e \in E_S$ from node A to node B ($v_0(e) = A$,$v_1(e) = B$) if and only if module B is created inside module A.

- A module cannot be created inside two different modules. There are no distinct edges $e_1, e_2 \in E_S$ such that $v_1(e_1) = v_1(e_2)$. This restriction makes the hierarchy of nested modules a tree. In fact, the subgraph $H|_S = \{N, E_S, v_0|_S, v_1|_S\}$ is a forest. The functions $v_0|_S$ and $v_1|_S$ are restrictions of v_0 and v_1 to E_S, respectively.

- The trees in $H|_S$ with only a single node represent the modules which have no other modules created inside them. The trees in $H|_S$ with more than one node represent

the hierarchies of nested modules: the root of the tree is the outermost module in the hierarchy and for each node A its parent represents the module in which A is created in and its children represent the modules created inside A. For a module A in a nested module hierarchy we define

$$inner(A) = \{X \mid \text{there is a path in } H|_S \text{ from } A \text{ to } X\}$$
$$outer(A) = \{X \mid \text{there is a path in } H|_S \text{ from } X \text{ to } A\}$$

in the equations above a path refers to a directed path.

- A module cannot import any of its inner modules and cannot import any of its outer modules. There is no edge $e \in E_I$ from B to A if $B \in inner(A)$ or $B \in outer(A)$.
- An import path in H is a path consisting only of edges from E_I and a nested path in H is a path consisting only of edges from E_S.
- A nested module A can only be imported by a module B if they have the same parent or if the parent of A is imported by the parent of the B. Conversely, A nested module A can only import a module B if they have the same parent or if the parent of A imports the parent of B. If none of two modules are a nested module they can import each other, respecting the restrictions imposed by the definition of H; a module cannot be imported twice by the same same module and, of course, there should be no cycles.
- More generally, for any two modules A and B an edge $e \in E_I$ such that $v_0(e) = B$ and $v_1(e) = A$ may exist in H only if
  - there is no edge $e_s \in E_S$ such that $v_1(e_s) = A$ or $v_1(e_s) = B$; or
  - there are edges $e_{s1}, e_{s2} \in E_S$ such that $v_1(e_{s1}) = B$, $v_1(e_{s2}) = A$, and $v_0(e_{s1}) = v_0(e_{s2})$; or
  - there are edges $e_{s1}, e_{s2} \in E_S$ and $e_{i1}, \ldots, e_{i2} \in E_I$ such that $v_1(e_{s1}) = B$, $v_1(e_{s2}) = A$, and $(e_{i1}, \ldots, e_{i2})$ is an import path from $v_0(e_{s1})$ to $v_0(e_{s2})$.

A.1.3 Using Objects

An object may be registered in any module having its class declared for object creation in its signature. An object cannot be registered in two different modules. However, an object may be used in any module having its class declared for object use, provided that the object is visible in that module.

- An object registered in a module B is visible in B and in any module A provided there is an import or a nested path from B to A.

A.2 TOOR Grammar and Automata

Regular expressions for object tracing are defined in terms of object and relation class identifiers. For a given configuration of objects and relations TOOR formalizes the set of all strings that can refer to actual objects and relations in terms of a configuration automaton. The search engine uses this automaton to find the strings matching the pattern described by a given regular expression. The actual match is found by computing the intersection of both the configuration automation and the automaton underlying the regular expression. The proposition below demonstrates the appropriateness of this approach.

Proposition 66 A string $w = s_0 s_1 \dots s_n$ conforms to a configuration state S if and only if w can be recognized by the corresponding configuration automaton A_S.

Proof: Consider that a string $w = s_0 s_1 \dots s_n$ conforms to $S = \{N, E, L, v_0, v_1, v_l\}$. We have, by definition of string conformance (Definition 60), that there are nodes $o_0, \dots, o_n$ in N such that

$$s_0 = o_0 \tag{A.1}$$

$$s_i = o_i, v_0(e) = o_{i-1}, v_1(e) = o_i, \text{ for some } e \in E; \text{ for } 0 = i \bmod 2,\ i = 2, \dots, n; \text{ and} \tag{A.2}$$

$$s_i = v_l(o_i), v_0(e) = o_{i-1}, v_1(e) = o_i, \text{ for some } e \in E; \text{ for } 1 = i \bmod 2,\ i = 1, \dots, n-1. \tag{A.3}$$

By definition of $A_S = (X, I, \delta, x_I, F)$ (Definition 63) we can take states $x_{o_0}, x_{o_1}, \dots, x_{o_n}$ from X_{obj}, one x_{o_i} for each o_i for $i = 0 \bmod 2,\ i = 2, \dots, n$, and states $\overline{x}_{o_0}, \overline{x}_{o_1}, \dots, \overline{x}_{o_n}$ from X_r, one $\overline{x}_{o_i}$ for each o_i for $1 = i \bmod 2,\ i = 1, \dots, n-1$. Now, from definition of δ and equations A.1, A.2, and A.3, it is easy to see that

$$\begin{aligned} &x_{o_0} \in \delta(x_I, s_0)\ ; \\ &x_{o_i} \in \delta(x_{o_{i-1}}, s_i) \text{ for } 0 = i \bmod 2, i = 2, \dots, n\text{; and} \\ &x_{o_i} \in \delta(x_{o_{i-1}}, s_i) \text{ for } 1 = i \bmod 2, i = 1, \dots, n-1. \end{aligned}$$

Therefore, $x_{o_0}, x_{o_1}, \dots, x_{o_n}$ form a sequence of states recognizing $w = s_0 s_1 \dots s_n$. The converse is proved similarly. For any $w = s_0, s_1, \dots, s_n$ recognized by A_S we show that the states $x_{o_i}, i = 0, \dots, n$, in the sequence of states $x_I, x_{o_0}, x_{o_1}, \dots, x_{o_n}$ recognizing w correspond to objects $o_0, \dots, o_n$ satisfying equations A.1, A.2, and A.3. □

A.3 Derivatives of Regular Expressions

Derivatives of regular expressions were introduced by Brzozowski [20] in 1964. This section presents the most important concepts to understand the use of derivatives in TOOR. However, further details should be sought in the original reference. We use the original notation of [20] to facilitate cross-reference. There is a clash with the notation we use in other parts of this thesis: the symbol δ is used to designate a characteristic function in terms of derivatives and to designate a state transition function when applied to automata. This should not be a problem since the derivate use of δ only occurs in this section.

- Given an alphabet A_k and a set R of strings (sequences of symbols from A_k) the derivative of R with respect to a symbol $s \in A_k$ is denoted by D_sR and defined as $D_sR = \{t \mid st \in R\}$.

- First we present the overall procedure based on derivatives. The necessary details will be given later on. The construction of an automaton recognizing the language denoted by a regular expression R over an alphabet $A_k = \{s_0, \ldots, s_n\}$ is described in the following way:

  1. Find the derivates of R with respect to strings of increasing length, i.e., $D_\lambda R$, $D_{s_0}R$, ..., $D_{s_n}R$, $D_{s_0s_0}R$, $D_{s_0s_1}R$, ...
  2. For each derivative D_wR of a different type associate a state X_{D_wR} of the automaton. For each derivative $D_{ws}R$ there is a transition under s from state X_{D_wR} to state $X_{D_{ws}R}$.
  3. The final states X_{D_wR} are those with $\delta(D_wR) = \lambda$.
  4. The procedure stops when for a given string of length n no new types of derivatives can be found.

- Reference [20] presents results showing that the construction above is accurate. Essentially it is shown that every regular expression may be described by a sum of derivatives, that for any regular expression there are only a limited number of different types of derivatives, and that a string w is in R if and only if $\lambda \in D_wR$.

- Two derivatives are said to be of the same type if they are equal (but not necessarily of the same form). We see from the above procedure that the essential aspects are the definition of δ and the calculation of D_sR for a regular expression R.

- In what follows we maintain the notation of [20]. P and Q are used for regular expressions, $*$ and $+$ for the regular operations of iteration and sum, respectively (the

concatenation of P and Q is represented by PQ), and $f(P,Q)$ for any boolean function of P and Q such as intersection, relative complement, exclusive or, etc. Also λ is the empty string and $\emptyset$ is the empty set. We use the same symbol for a string s and for the set consisting of only of s.

- Given a set of strings R, $\delta(R)$ is defined as

$$\delta(R) = \begin{cases} \lambda & \text{if} \quad \lambda \in R \\ \emptyset & \text{if} \quad \lambda \notin R \end{cases}$$

where, λ is the set consisting only of the empty string and $\emptyset$ is the empty set.

- Derivatives of regular expressions with respect to a symbol $a \in A_k$ are defined recursively as

$$\begin{aligned} D_a a &= \lambda \\ D_a b &= \emptyset \text{ for } b = \lambda, \text{ or } b = \emptyset, \text{ or } b \in A_k \text{ and } b \neq a \\ D_a(P*) &= D_a(P)P* \\ D_a(PQ) &= D_a(P)Q + \delta(P)D_a(Q) \\ D_a(f(P,Q)) &= f(D_a(P), D_a(Q)) \end{aligned}$$

where $f(P,Q)$ is a boolean function of P and Q.

- Derivatives of R are extended from symbols to strings: $D_{ws}R = D_s(D_wR)$ and $D_\lambda R = R$.

A.4 Derivatives of TOOR Regular Definitions

The use of derivatives of regular expressions in TOOR is not straightforward because the various regular definitions characterizing identifiers, classes, and object properties are not symbols of the alphabet of object and relation class identifiers ($\mathcal{I}_{obj} \cup \mathcal{I}_{rc}$). Regular definitions in TOOR are regular expressions defined in terms of the identifier alphabet and axioms. We define what is meant by a derivative of a regular definition with respect to a symbol from the alphabet of object and relation class identifiers. This is done in two steps. First, we characterize regular definitions as regular expressions over $A = \mathcal{I}_{obj} \cup \mathcal{I}_{rc}$. Second, we present the definition of a derivate for each regular definition characterized in such a way.

- We use the word `regid` to denote a regular expression over the identifier alphabet and the word `regdef` to denote any of the regular definitions `<regid>`, `[regid]`, `!regid!`, or the regular definition for properties `[[C x if` $\langle FOOPSaxiom \rangle$`]]`.

- For every regular expression `regid` used in a regular definition, $match(s, \texttt{regid})$ is true if the string s matches the pattern `regid` and false otherwise. Also, for regular definitions `[[regaxm]]` where `regaxm` is an expression of the form `C x if` $\langle FOOPSaxiom \rangle$ we have that $match(s, \texttt{regaxm})$ is true if the object identified by s is of class C and the axiom is true of s, and $match(s, \texttt{regaxm})$ is false otherwise. For any object identifier o_i, $class(o_i)$ returns the class of o_i.

- A regular definition `<regid>` for object identifier is matched by any object o_i such that o_i matches the pattern `regid`.

$$\texttt{<regid>} \;=\; (o_1 \mid \ldots \mid o_n) \quad \text{for } o_i \in \mathcal{I}_{obj} \text{ and } match(o_i, \texttt{regid}) = \text{true}.$$

- A regular definition `[regid]` for class identifier is matched by any object o_i such that its class matches the pattern `defc`.

$$\texttt{[regid]} \;=\; (o_1 \mid \ldots \mid o_n) \quad \text{for } o_i \in \mathcal{I}_{obj} \text{ and } match(class(o_i), \texttt{regid}) = \text{true}.$$

- A regular definition `[[C : x if` $\langle FOOPSaxiom \rangle$`]]` is matched by any object o_i of class `C` such that the $\langle FOOPSaxiom \rangle$ applied to o_i is true.

$$\texttt{[[regaxm]]} \;=\; (o_1 \mid \ldots \mid o_n) \quad \text{for } o_i \in \mathcal{I}_{obj} \text{ and } match(o_i, \texttt{regaxm}) = \text{true}.$$

- A regular defnition `!regid!` for negation is matched by any object o_i such that its class does not match `defneg`.

$$\texttt{!regid!} \;=\; (o_1 \mid \ldots \mid o_n) \quad \text{for } o_i \in \mathcal{I}_{obj} \text{ and } match(class(o_i), \texttt{regid}) = \text{false}.$$

- For every regular definition `regdef` we define `regdef`$\triangleleft C \triangleright$ to be the difference between the set denoted by `regdef` and the set C of strings. The set denoted by `regdef` may itself be expressed as `regdef`$\triangleleft \emptyset \triangleright$.

- The derivative of a regular definition `regdef`$\triangleleft C \triangleright$ with respect to a symbol $s \in \mathcal{I}_{obj} \cup \mathcal{I}_{rc}$ is defined by

$$D_s(\texttt{<regid>}\triangleleft C\triangleright) = \begin{cases} \texttt{<regid>}\triangleleft C \cup \{s\}\triangleright & \text{if } s \in \mathcal{I}_{obj} \text{ and } s \notin C \text{ and} \\ & match(s, \texttt{regid}) = \text{true} \\ \emptyset & \text{otherwise} \end{cases}$$

$$D_s(\texttt{[regid]}\triangleleft C\triangleright) = \begin{cases} \texttt{[regid]}\triangleleft C \cup \{s\}\triangleright & \text{if } s \in \mathcal{I}_{obj} \text{ and } s \notin C \text{ and} \\ & match(class(s), \texttt{regid}) = \text{true} \\ \emptyset & \text{otherwise} \end{cases}$$

$$D_s(\texttt{[[regaxm]]}\triangleleft C\triangleright) = \begin{cases} \texttt{[[regaxm]]}\triangleleft C \cup \{s\}\triangleright & \text{if } s \in \mathcal{I}_{obj} \text{ and } s \notin C \text{ and} \\ & match(s, \texttt{regaxm}) = \text{true} \\ \emptyset & \text{otherwise} \end{cases}$$

$$D_s(\texttt{!regid!}\triangleleft C\triangleright) = \begin{cases} \texttt{!regid!}\triangleleft C \cup \{s\}\triangleright & \text{if } s \in \mathcal{I}_{obj} \text{ and } s \notin C \text{ and} \\ & match(class(s), \texttt{regid}) = \text{false} \\ \emptyset & \text{otherwise} \end{cases}$$

- For any regular definition $\texttt{regdef}\triangleleft C\triangleright$ we define

$$\delta(\texttt{regdef}\triangleleft C\triangleright) = \begin{cases} \lambda & \text{if} \quad C \neq \emptyset \\ \emptyset & \text{if} \quad C = \emptyset \end{cases}$$

- The negation operator is defined as the complement relative to a universe of strings I. For a given configuration state with $\mathcal{I}_{obj} = \{o_1, \ldots, o_n\}$ and $\mathcal{I}_{rc} = \{r_1, \ldots, r_m\}$, we take the universe I of strings in TOOR to be denoted by the regular expression $(o_1 \mid \ldots \mid o_n \mid r_1 \mid \ldots \mid r_m)*$. This definition is accurate in the sense that every valid string, according to the configuration grammar, is contained in I. On the other hand, I contains more strings than the valid ones. However, this is not a problem because the result of computing derivatives of I is expressed in terms of I which ultimately will be matched only by the actual strings in TOOR. Also, expressing I in this way makes the calculation of derivatives of the universe very simple. For any symbol $s \in \mathcal{I}_{obj} \cup \mathcal{I}_{rc}$ we have $D_s I = I$.

Bibliography

[1] Alfred V. Aho. Pattern matching in strings. In R.V. Book, editor, *Formal Language Theory*, pages 325–347, New York, 1980. Academic Press.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[4] Mack W. Alford. A requirements engineering methodology for real-time processing requirements. *IEEE Transactions on Software Engineering*, SE-3(1):60–69, January 1977.

[5] Mack W. Alford. Requirements for distributed data processing design. In *First International Conference on Distributed Computing Systems*, pages 1–14, Washington D.C., 1979. Computer Society Press of the IEEE.

[6] Mack W. Alford. Software requirements in the 80's: From alchemy to science. In *Proceedings of the ACM '80 Conference*, pages 342–349. ACM press, 1980.

[7] Mack W. Alford. SREM at the age of eight; the distributed computing design system. *IEEE Computer*, 18(4):36–46, April 1985.

[8] Mack W. Alford and I.F. Burns. R-nets: A graph model for real-time processing requirements. In *Symposium on Computer Software Engineering*, pages 97–108, New York, 1976. Polytechnic Press.

[9] Thomas E. Bell, David C. Bixler, and Margaret E. Dyer. An extendable approach to computer-aided software requirements engineering. *IEEE Transactions on Software Engineering*, 3(1):49–60, January 1977.

[10] Jan Bergstra, Jan Heering, and Paul Klink. Module algebra. *Journal of the ACM*, 37(2):335–372, 1990.

[11] Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

[12] Edward H. Bersoff and Alan M. Davis. Impacts of life cycle models on software. *Communications of the ACM*, 34(8):104–117, August 1991.

[13] Barry W. Boehm. Software engineering. *IEEE Transactions on Computers*, 25(12):1226–1241, December 1976.

[14] Barry W. Boehm. Software life cycle factors. In C.R. Vick and C.V. Ramamoorthy, editors, *Handbook of Software Engineering*, pages 494–518, New York, 1984. Van Nostrand-Reinhold.

[15] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, January 1984.

[16] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[17] A. Brown and J. McDermid. On integration and re-use in a software development environment. In F. Long, editor, *Software Engineering Environments*, volume 3. Ellis Hoorwood, 1991.

[18] A. Brown and J. McDermid. Learning from IPSE mistakes. *IEEE Software*, 9(2), 1992.

[19] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120:197–213, 1993.

[20] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[21] J. Bubenko. Keynote address. In *Proceedings, Second IEEE International Symposium on Requirements Engineering*, York, England, March 1995. IEEE Computer Society Press.

[22] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058, Department of Computer Science, Carnegie-Mellon University, 1977.

[23] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification,* Lecture Notes on Computer Science, volume 86, pages 292–332. Springer, 1980. Based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warsaw, Poland, 1978.

[24] Graham Button and Wes Sharrock. Occasioned practices in the work of software engineers. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, Computers and People, pages 217–240, London, UK, 1994. Academic Press Limited.

[25] Jeff Conklin and Michael L. Begeman. gIBIS: A hypertext tool for exploratory policy discussion. *ACM Transactions on Office Information Systems*, 6(4):303–331, October 1988.

[26] Alan M. Davis. Automating the requirements phase: Benefits to later phases of the software life cycle. In *IEEE COMPSAC'80*, pages 42–48, Washington, D.C., October 1980. IEEE Computer Society Press.

[27] Alan M. Davis. The design of a family of application-oriented requirements languages. *IEEE Computer*, Find(out):21–28, May 1982.

[28] Alan M. Davis. Operational prototyping: A new development approach. *IEEE Software*, 9(5):70–78, September 1992.

[29] Alan M. Davis, Edward H. Bersoff, and Edward R. Comer. A strategy for comparing alternative software development life cycle models. *IEEE Transactions on Software Engineering*, 14(10):1453–1461, October 1988.

[30] Alan M. Davis and Others. RLP: An automated tool for the automatic processing of requirements. In *IEEE COMPSAC'79*, pages 188–194, Washington, D.C., November 1979. IEEE Computer Society Press. Also, GTE Automatic Electric Worldwide Communications Journal, 17(6):289-299, 1979 .

[31] Alan M. Davis and I. Rataj. Requirements language processing for the effective testing of real-time systems. In *ACM SIGSOFT Software Engineering Notes*, volume 3(5), pages 61–66, November 1978.

[32] Alan M. Davis and T. G. Rauscher. Formal techniques and automatic processing to ensure correctnes in requirements specifications. In *Proceedings, Specifications of Reliable Software Conference*, pages 15–35, Los Alamitos,Calif., 1979. IEEE Computer Society Press.

[33] Alan Michael Davis. *Software requirements : objects, functions and states.* Prentice-Hall, Englewood Cliffs, London, 1993.

[34] Carl G. Davis and Charles R. Vick. The software development system. *IEEE Transactions on Software Engineering*, 3(1):69–84, January 1977.

[35] C. W. Dawson and R. J. Dawson. Towards more flexible management of software systems development using meta-models. *Software Engineering Journal*, 11(3):79–88, May 1995.

[36] Tom DeMarco. *Structured Analysis and System Specification.* Prentice-Hall, Englewood Cliffs, London, reprint edition, 1979.

[37] Răzvan Diaconescu. Category-based semantics for equational and constraint logic programming. Technical Monograph PRG-116, Oxford University Computing Laboratory, July 1994.

[38] Merlin Dorfman and R. F. Flynn. ARTS - an automated requirements traceability system. *Journal of Systems and Software*, 4(1):63–74, 1984.

[39] Ken Eason and Wendy Olphert. Early evaluation of the organisational implications of CSCW systems. In Peter Thomas, editor, *CSCW Requirements and Evaluation*, Computer Supported Cooperative Work, pages 75–89. BCS - Springer-Verlag, 1996.

[40] Steve Easterbrook. Negotiation and the role of the requirements specification. Technical Report Cognitive science research paper CSRP 197, School of Cognitive and Computing Sciences, University of Sussex, Brighton,UK, 1991.

[41] Steve Easterbrook and Bashar Nuseibeh. Managing inconsistencies in an evolving specification. Technical Report to find, School of Cognitive and Computing Sciences, University of Sussex, Brighton,UK, 1994.

[42] Robert W. Ellis. An ethnographic study of requirements elicitation practices at a commercial requirements engineering consultancy. Master's thesis, Oxford University Computing Laboratory, September 1995.

[43] Stuart I. Feldman. MAKE – a program for maintaining computer programs. *Software, Practice and Experience*, 9(3):255–265, March 1979.

[44] A Finkelstein, M Goedicke, J Kramer, and C Niskier. Viewpoint oriented software-development - methods and viewpoints in requirements engineering. *Lecture Notes in Computer Science*, 490:29–54, 1991.

[45] Anthony Finkelstein, D. Gabbay, A. Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multi-perspective specifications. *IEEE Transactions on Software Engineering*, 20(8), August 1994.

[46] Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press / Willey, 1995.

[47] R. Flynn and M. Dorfman. The automated requirements traceability system (ARTS). In R. Thayer and M. Dorfman, editors, *System and Software Requirements Enginnering*, pages 423–438. IEEE Computer Society Press, Washington, D.C., 1990.

[48] V.M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

[49] Joseph Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, 10(5):528–543, September 1984.

[50] Joseph Goguen. Suggestions for using and organizing libraries in software development. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, First International Conference on Supercomputing Systems*, pages 349–360. IEEE Computer Society, 1985. Also in *Supercomputing Systems*, Steven and Svetlana Kartashev, Eds., Elsevier, 1986.

[51] Joseph Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, February 1986.

[52] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison-Wesley, 1989.

[53] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology*, Joint System Developmen Corporation. Tokyo, Japan, January 1990.

[54] Joseph Goguen. The dry and the wet. In Eckhard Falkenberg, Colette Rolland, and El-Sayed Nasr-El-Dein El-Sayed, editors, *Information Systems Concepts*, pages 1–17. Elsevier North-Holland, 1992. Proceedings, IFIP Working Group 8.1 Conference (Alexandria, Egypt); Also, Technical Monograph PRG-100, Centre for Requirements and Foundations, Oxford University Computing Laboratory, 1992.

[55] Joseph Goguen. Requirements engineering as the reconciliation of social and technical issues. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 165–199. Academic Press, 1994.

[56] Joseph Goguen. Parameterized programming and software architecture. 1996. To appear.

[57] Joseph Goguen. Towards a social theory of information. In Leigh Star, Geoffrey Bowker, and William Turner, editors, *Workshop on Social Science Research, Technical Systems and Cooperative Work*, (to appear) 1994. Also, Technical report, Centre for Requirements and Foundations, Oxford University Computing Laboratory, 1993.

[58] Joseph Goguen. Towards a social theory of information. In Geoffrey Bowker, Les Gasser, Leigh Star, and William Turner, editors, *Social Science Research: Technical Systems and Cooperative Work*. Erlbaum, (to appear) 1996.

[59] Joseph Goguen and Rod Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report CSL-118, SRI Computer Science Laboratory, 1980.

[60] Joseph Goguen and Charlotte Linde. Techniques for requirement elicitation. In Stephen Fickas and Anthony Finkelstein, editors, *Proceedings, IEEE International Symposium on Requirements Engineering*, pages 152–164, California, January 1993. Also, Technical report, Centre for Requirements and Foundations, Oxford University Computing Laboratory, 1992.

[61] Joseph Goguen and José Messeguer. Unifying functional, object-oriented, and relational programming with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986.

[62] Joseph A. Goguen. Formality and informality in requirements engineering. In *4th International Conference on Requirements Engineering*, pages 102–108. IEEE Computer Society Press, April 1996.

[63] Orlena Gotel. *Contribution Structures for Requirements Traceability*. PhD thesis, Department of Computing, Imperial College of Science, Technology, and Medicine, University of London, August 1995.

[64] Orlena Gotel and Anthony Finkelstein. An analysis of the requirements traceability problem. Technical Report TR-93-41, Department of Computing, Imperial College of Science, Technology and Medicine, 1993.

[65] Orlena Gotel and Anthony Finkelstein. Contribution structures. In *Proceedings, Second IEEE International Symposium on Requirements Engineering*, pages 100–107. IEEE Computer Society Press, March 1995.

[66] Orlena Gotel and Anthony Finkelstein. Revisiting requirements production. *Software Engineering Journal*, 11, 1996.

[67] Sol Greenspan, John Mylopoulos, and Alex Borgida. On formal requirements modelling languages: RML revisited. In *Proceedings, Sixteenth International Conference on Software Engineering*, pages 135–147, Sorrento,Italy, May 1994. IEEE Computer Society Press. Invited Plenary Talk.

[68] Gary Griffiths. CASE in the third generation. *Software Engineering Journal*, 9(4):159–166, July 1994.

[69] M. Hamilton and S. Zeldin. The functional life cycle model and its automation: USE.IT. *Journal of Systems and Software*, 3(1):25–62, March 1983.

[70] Hubert F. Hofmann, Rolf Pfeifer, and Erik Vinkhuyzen. Situated software design. Technical Report ifi-93.18, Computer Science Department, University of Zurich, 1993.

[71] E. Horowitz and R.C. Williamson. SODOS: a software documentation support environment - its definition. *IEEE Transactions on Software Engineering*, 12(8):849–859, 1986.

[72] Ellis Horowitz and Ronald C. Williamson. SODOS: A software documentation support environment – its use. *IEEE Transactions on Software Engineering*, 12(11):1076–1087, November 1986.

[73] John Hughes, Jon O'Brien, Tom Rodden, Mark Rouncefield, and Ian Sommerville. Presenting ethnography in the requirements process. Technical Report SE/5/1994, Software Engineering Research Group, Computing Department, Lancaster University, Lancaster,UK, 1994.

[74] INCOSE. Requirements management tools survey. Presented at the International Council on Systems Engineering symposium – Tools Database Working Group, 1996. available via http: //www.incose.org/workgrps/tools/ tooltax.html.

[75] Michael A. Jackson. *System Development.* Prentice-Hall International Englewood Cliffs, NJ, 1983.

[76] Michael A. Jackson and Ralph Storer. *Practical Program Development Using JSP.* Computer science texts. Blackwell Scientific, Oxford, 1987.

[77] M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, and Y. Vassiliou. Theories underlying requirements engineering: An overview of NATURE at genesis. In Stephen Fickas and Anthony Finkelstein, editors, *Proceedings, IEEE International Symposium on Requirements Engineering*, pages 19–33, California, January 1993.

[78] M. Jarke, H. Nissen, and Klaus Pohl. Tool integration in evolving information systems environments. In *3rd GI Workshop Information Systems and Artificial Intelligence: Administration and Processing of Complex Structures*, Hamburg, Germany, February 1994.

[79] H. Kaindl. The missing link in requirements engineering. *ACM SIGSOFT Software Engineering Notes*, 18(2):30–39, 1993.

[80] H. Kawashima et al. Functional specification of call processing by state transition diagram. *IEEE Transactions on Communications*, 19(5):581–587, October 1971.

[81] D. A. Khun. Automating systems engineering. Master's thesis, University of Texas, November 1993.

[82] Rudolf J. Lauber. Development support systems. *IEEE Computer*, 15(5):36–46, May 1982.

[83] Laurence James. Current trends in requirements & specification management. Marconi Systems Technology internal report, June 1995. available via http://www.mstus.com/ trends.htm.

[84] Laurence James. Using RTM for automotive safety-related software development. Marconi Systems Technology internal report, 1995. available via http://www.mstus.com/ safety.htm.

[85] M. M. Lehman. Programs, life cycles, and laws of software evolution. In *Proceedings of IEEE*, volume 68, pages 1060–1076, 1980.

[86] Michael Löwe, Hartmut Ehrig, Werner Fey, and Dean Jacobs. On the relationship between algebraic module specifications and program modules. In Abramsky S and T.S.E. Maibaum, editors, *Proceedings, International Conference on Theory and Practice of Software Development*, 1991. Lecture Notes on Computer Science, 494(2), 83-89.

[87] Luqi. Software evolution through rapid prototyping. *IEEE Computer*, 22(5):13–25, May 1989.

[88] Luqi and Joseph Goguen. Formal methods: Promisses and problems. *IEEE Software*, 1996. To appear.

[89] Luqi and Mohammad Ketabchi. A computer-aided prototyping system. *IEEE Software*, 5(2):66–72, March 1988.

[90] Luqi and Robert Steigerwald. CAPS as a requirements engineering tool. *Communications of the ACM*, pages 75–83, 1991.

[91] Nadine Malcolm and Mack Alford. Systems engineering: Its roots, its future. Ascent Logic Corporation internal report, 180 Rose Orchad Way, San Jose, CA 95134, 1995. available via http://www.alc.com/.

[92] Nadine Malcolm and Jane McKennon. Integrating a concurrent engineering tool set: Issues and approaches. Ascent Logic Corporation internal report, 180 Rose Orchad Way, San Jose, CA 95134, 1995. available via http://www.alc.com/integrating.html.

[93] D.D. McCracken and M.A. Jackson. Life-cycle concept considered harmful. *ACM SIGSOFT Software Engineering Notes*, 7(2):29–32, April 1982.

[94] John A. McDermid. Requirements analysis: Orthodoxy, fundamentalism and heresy. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, Computers and People, pages 17–40, London, UK, 1994. Academic Press Limited.

[95] Simon McGinnes. CASE support for collaborative modelling: Re-engineering conceptual modelling techniques to exploit the potential of CASE tools. *Software Engineering Journal*, 1994.

[96] MST. Requirements and traceability management. Marconi Systems Technology internal report 300/HD/01553/002. available via http://www.mstus.com/prodescr.htm.

[97] James A.H. Murray, Henry Bradley, W.A. Graigie, and C.T. Onions, editors. *The Oxford English Dictionary*. Oxford University Press, Oxford, UK, 1989.

[98] NASA. MO & DSD tools capabilities inventory. Report on tool's demonstration made by vendors, 1996. available via http: //joy.gfsc.nasa.gov/MSEE/.

[99] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, October 1994.

[100] Institute of Electric and Eletronics Engineers. *IEEE Guide to Software Requirements Specifications*. IEEE/ANSI Standard 830-1984, New York, 1984.

[101] Hajime Ohiwa, Kazuhisa Kawai, and Masanobu Koyama. Idea processor and the KJ method. *Journal of Information Processing*, 13(1):44–48, 1990.

[102] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[103] David Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[104] David Parnas. On a 'buzzword': Hierarchical structure. *Information Processing*, 74:336–339, 1974.

[105] David Parnas. The use of precise specifications in the development of software. *Information Processing*, 77:861–867, 1977.

[106] Lawrence Peters. Relating software requirements and design. In *Proceedings of the Software Quality and Assurance Workshop*, pages 67–71, San Diego, CA, November 1978. Published in a special joint issue of ACM Software Engineering Notes, 3(5), and ACM Performance Evaluation Review, 7(3 & 4); 1978.

[107] J. Peterson and Yngvar D. Tronstad. An RDD based forward and reverse engineering design process. In *Third Annual Re-engineering Technoly Workshop*. NSWC, August 1992. available via http://www.alc.com/reveng.html.

[108] Robert A. Pierce. A requirements tracing tool. *ACM SIGSOFT Software Engineering Notes*, 3(5):53–60, November 1978.

[109] Francisco A.C. Pinheiro and Joseph A. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, March 1996.

[110] Klaus Pohl and S. Jacobs. Traceability between cross-functional-teams. In *1st International Conference on Concurrent Engineering, Research and Application*, Pittsburgh, USA, August 1994.

[111] Klaus Pohl and Peter Peters. Workshop summary – second international workshop on requirements engineering: Foundations of software quality (REFSQ'95). *ACM SIGSOFT Software Engineering Notes*, 21(1):31–34, January 1996.

[112] C. Potts and G. Burns. Recording reasons for design decisions. In *Proceedings, 10th International Conference on Software Engineering*, pages 418–426, April 1988.

[113] QSS. DOORS, management through requirements. Promotional material, Quality Systems and Software Ltd. Magdalen Centre, Oxford Science Park, Oxford, OX4 4GA, 1993.

[114] QSS. Requirements and DOORS – questions and answers about DOORS version 2. Promotial booklet, Quality Systems and Software Ltd. Magdalen Centre, Oxford Science Park, Oxford, OX4 4GA, 1993.

[115] QSS. Structured requirements training course. Course material, Quality Systems and Software Ltd. Magdalen Centre, Oxford Science Park, Oxford, OX4 4GA, March 1994.

[116] C.V. Ramamoorthy, Atul Prakash, W.T. Tsai, and Yutaka Usuda. Software engineering: Problems and perspectives. *IEEE Computer*, pages 191–209, October 1986.

[117] C.V. Ramamoorthy, Yutaka Usuda, Atul Prakash, and W.T. Tsai. The evolution support environment system. *IEEE Transactions on Software Engineering*, 16(11):1225–1234, November 1990. .

[118] Balasubramaniam Ramesh and V. Dhar. Supporting systems development using knowledge captured during requirements engineering. *IEEE Transactions on Software Engineering*, 18(6):498–510, June 1992.

[119] Balasubramaniam Ramesh and V. Dhar. Representing and maintaining process knowledge for large scale systems development. *IEEE Expert*, 9(2):54–59, 1994.

[120] Balasubramaniam Ramesh and Luqi. Process knowledge based rapid prototyping for requirements engineering. In Stephen Fickas and Anthony Finkelstein, editors, *Proceedings, IEEE International Symposium on Requirements Engineering*, pages 248–255, California, January 1993.

[121] Balasubramaniam Ramesh and Luqi. An intelligent assistant for requirements validation for embedded systems. Submitted to *IEEE Expert*, 1994.

[122] Dave Randall, Michael Twindale, and Richard Bentley. Dealing with uncertainty – perspectives on the evaluation process. In Peter Thomas, editor, *CSCW Requirements and Evaluation*, Computer Supported Cooperative Work, pages 141–155. BCS - Springer-Verlag, 1996.

[123] Lucia Rapanotti and Adolfo Socorro. Introducing FOOPS. Technical Report PRG-TR-28-92, Programming Research Group, Oxford University Computing Laboratory, 1992.

[124] Gail L. Rein and Clarence A. Ellis. rIBIS: a real-time group hypertext system. *International Journal of Man-Machine Studies*, 34(3):349–367, March 1991.

[125] A. Rockstrom and R. Saracco. SDL-CCITT specification and description language. *IEEE Transactions on Communications*, 30(6):1310–1318, June 1982.

[126] W. Royce. Managing the development of large software systems. In *IEEE WESCON*, pages 1–9, August 1970. Also, reprinted in 9th IEEE International Conference on Software Engineering, Washington, D.C.: IEEE Computer Society Press, 1987, pp 328-338.

[127] Stephen A. R. Scrivener, Silvia Pongutá Urquijo, and Hilary K. Palmén. The use of breakdown analysis in synchronous CSCW system design. In Peter Thomas, editor, *CSCW Requirements and Evaluation*, Computer Supported Cooperative Work, pages 157–172. BCS - Springer-Verlag, 1996.

[128] Gareth Smith and Tom Rodden. Using shared interface objects to support cooperation. Technical Report CSCW/2/1994, Centre for Research in CSCW, Computing Department, Lancaster University, Lancaster,UK, 1994.

[129] Adolfo Socorro. *Design, Implementation and Evaluation of a Declarative Object-Oriented Programming Language.* PhD thesis, Oxford University Computing Laboratory, 1993.

[130] Ian Sommerville. *Software Engineering.* Addison-Wesley, 1989.

[131] Ian Sommerville and Tom Rodden. Requirements engineering for cooperative systems. Technical Report CSCW/1/1994, Centre for Research in CSCW, Computing Department, Lancaster University, Lancaster,UK, 1994.

[132] Richard Stevens and James Martin. What is requirements management? QSS internal report, Quality Systems and Software Ltd. Magdalen Centre, Oxford Science Park, Oxford, OX4 4GA, 1996. available via http://www.qss.co.uk/reqsmgmt.html.

[133] Lucy Suchman. *Plans and Situated Actions.* Cambridge University Press, 1987.

[134] Naohiko Takeda, Akichika Shiomi, Kazuhisa Kawai, and Hajime Ohiwa. Requirement analysis by the KJ editor. In Stephen Fickas and Anthony Finkelstein, editors, *Proceedings, IEEE International Symposium on Requirements Engineering*, pages 98–101, California, January 1993.

[135] B. J. Taylor. A methodology for expressing the functional requirements of real-time systems. In *IFAC/IFIP Workshop on Real-Time Programming*, pages 111–120, Leibnitz, Austria, 1980.

[136] Daniel Teichroew and Ernest A. Hershey. PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems. *IEEE Transactions on Software Engineering*, 3(1):41–48, January 1977.

[137] Daniel Teichroew and H. Sayani. Automation of system building. *Datamation*, pages 25–30, August 1971.

[138] Daniel Teichroew and H. Sayani. Computer-aided requirements engineering. In *ACM'80 Conference*, pages 369–381, New York, 1980. ACM Press.

[139] Design Theory, Computer Science: processes, and methodology of computer systems design. *Subrata Dasgupta.* Number 15 in Cambridge tracts in theoretical computer science. Cambridge University Press, 1991.

[140] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[141] Dimitrios Tombros and Andreas Geppert. A survey of database support for process-centered software development environments. Technical Report ifi-95-28, Computer Science Department, University of Zurich, 1995.

[142] Yngvar D. Tronstad, Robert Earley, Dennis Biddle, and Jack Ashburner. Information system reengineering process: A system approach. Ascent Logic Corporation white paper, 180 Rose Orchad Way, San Jose, CA 95134, 1995. available via http://www.alc.com.

[143] Axel van Lamsweerde, Robert Darimont, and Philippe Massonet. Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt. In *Proceedings, Second IEEE International Symposium on Requirements Engineering*, pages 194–203, York, England, March 1995. IEEE Computer Society Press.

[144] C. Walker. The applicability of RTM to the capability maturity model and practices. Marconi Systems Technology internal report, 1995. available via http://www.mstus.com/cmm.htm.

[145] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward megaprogramming. *Communications of the ACM*, 35(11):89–99, 1992.

[146] Roel J. Wieringa. An introduction to requirements traceability. Technical Report IR-389, Faculty of Mathematics and Computer Science, University of Vrije, Amsterdam, September 1995.

[147] R. Williams. Managing the development of reliable software. In *International Conference on Reliable Software*, April 1977.

[148] R. T. Yeh. System development as a wicked problem. *International Journal of Software Engineering and Knowledge Engineering*, 1(2):117–130, 1991.

[149] Raymond T. Yeh, David A. Naumann, Roland T. Mittermeir, Reinhard A. Schlemmer, William S. Gilmore, George E. Sumrall, and John T. Lebaron. A commonsense management model. *IEEE Software*, 8(6):23–33, November 1991.

[150] Weider D. Yu. Verifying software requirements - a requirement tracing methodology and its software tool - RADIX. *IEEE Journal on Selected Areas in Communications*, 12(2):234–240, 1994.