Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems

Dag I. K. Sjøberg

A thesis submitted to the Faculty of Science,
University of Glasgow
for the degree of Doctor of Philosophy
July 1993

Thesis
9595
Copy 1

Abstract

The research presented in this thesis establishes thesauri as a viable foundation for models, methodologies and tools for change management. Most of the research has been undertaken in a persistent programming environment. Persistent language technology has enabled the construction of sophisticated and well-integrated change management tools; tools and applications reside in the same store. At the same time, the research has enhanced persistent programming environments with models, methodologies and tools that are crucial to the exploitation of persistent programming in construction and maintenance of long-lived, data-intensive application systems.

Application programming deals with a very high rate of change to data definitions, dependent programs and dependent user interfaces. This leads to severe problems in propagating changes correctly. It is common to find that necessary changes consequent on some other change have not been made, so that the system is inconsistent and will *eventually* fail to operate correctly. This thesis documents the problems by reporting an industrial experiment that quantified and helped solve such problems. The major component of the experiment was the HMS thesaurus tool which automatically generates and updates name and identifier information in all the software written in all the languages of the application system.

It is demonstrated that a similar thesaurus tool in a persistent programming environment can serve as a basis for models and methodologies for building and maintaining large and long-lived application systems. An example is SPASM which is a set of constraints that should be adhered to in order to prevent deteriorating structure and improve maintainability.

The EnvMake tool automatically verifies programs according to the SPASM constraints. It also supports other steps of the introduced construction and maintenance methodology. In large-scale application development, build management such as installation, recompilation, relinking and re-execution is time consuming and error-prone without appropriate tool support. EnvMake automatically tracks down dependencies and initiates the appropriate actions.

Acknowledgements

First of all I am indebted to my supervisor Malcolm Atkinson for his continuous guidance, encouragement, enthusiasm and inexhaustible source of ideas. The support, whatever the issue, is very much appreciated. I am also grateful to my second supervisor Ray Welland for his advice and support. Malcolm and Ray carefully reviewed this thesis and made up a superb supervisory team providing, among other things, many stimulating discussions.

Numerous discussions with Paul Philbrow, Phil Trinder, João Lopes, Richard Cooper and other colleagues at Glasgow have helped the understanding of the issues of this research. I gratefully acknowledge the patient proof-reading help provided by Anne Philbrow. I particularly thank Mark Dunlop for his generous and artistic help, at a moment of crisis, with Figure 1.1.

The St Andrews persistent programming team, led by Ron Morrison, made many useful comments and influenced the work in several ways. Quintin Cutts and Graham Kirby provided excellent software for me to reuse. Quintin's assistance at the early stages of this research was invaluable.

Thanks also to many people in academia and industry in the UK and Norway who have contributed to the research presented in this thesis. In particular, the generosity of Tim King of PSL, in permitting three visits to PSL and 18 months access to project data, is very much acknowledged.

The extensive financial support by the Research Council of Norway, Division NAVF in terms of a postgraduate scholarship made this research possible. Some of the work has been conducted in the context of $FIDE_1$ (ESPRIT Basic Research Action 3070) and $FIDE_2$ (ESPRIT Basic Research Action 6309) research projects.

Table of Contents

Chapter 1: Introduction 1

1.1 The Problem of Change 3

1.2 Software Maintenance 4

1.3 Thesauri as Foundation for Change Management Tools 5

1.4 A Thesaurus Tool in an Industrial Environment 6

1.4.1 Schema Evolution Measurements 6

1.5 Thesaurus-Based Tools in Persistent Environments 7

1.5.1 A Persistent Thesaurus Tool 9

1.5.2 Models and Methodologies 10

1.5.3 EnvMake – Another Thesaurus-Based Supporting Tool 10

1.6 Thesis Statement 11

1.7 Thesis Structure 11

Chapter 2: The HMS Thesaurus Tool – An Industrial Experiment 13

2.1 Introduction 13

2.1.1 The HMS System 14

2.2 The Thesaurus Tool 15

2.2.1 The Meta-Data Relations 16

2.2.1.1 The Thesaurus Relation 16

2.2.1.2 The Query_Dictionary Relation 19

2.2.1.3 The Versions_Thesaurus Relation 20

2.2.2 The Thesaurus Interface 20

2.2.2.1 Name Usage Information 21

2.2.2.2 Schema Evolution – Impact Analysis 22

2.2.2.3 Consistency Checks 24

2.2.2.4 Change History 24

2.2.3 Implementation 25

2.2.4 Evaluation 28

2.2.4.1 Detecting Inconsistencies 28

2.2.4.2 Software Reuse 28

2.2.4.3 Performance 29

2.2.4.4 Learning and Understanding 29

2.2.4.5 Alternatives to the Tool 30

2.2.4.6 Granularity of Container Types 31

2.2.4.7 Recording Change 31

2.2.4.8 The Tool in an Organisational Context 32

2.3 Quantifying Evolution 33

2.3.1 Evolution of the HMS Schema 34

2.3.2 Consequences of the Schema Evolution 36
2.3.3 Problems of Measuring Evolution 40
2.3.4 Schema Evolution in Different Application Domains 42
2.4 Summary 43

Chapter 3: Software Evolution and Supporting Tools – A Survey 45

3.1 Introduction 45
3.1.1 The Software Development and Maintenance Process 45
3.1.1.1 Data Modelling 48
3.1.1.2 Formal Specifications 48
3.1.1.3 Automatic Documentation 48
3.1.1.4 Reverse Engineering 48
3.1.2 Change Management – An Aspect of Project Management 49
3.1.2.1 Software Process Modelling 49
3.1.3 Software Change Management – Focus of this Thesis 50
3.2 Schema and Type Evolution 51
3.2.1 Consequences on other Parts of Schema 52
3.2.2 Consequences on Extensional Data 53
3.2.2.1 Conversion 53
3.2.2.2 Filtering 53
3.2.3 Consequences on Application Programs 54
3.2.4 Approaches 54
3.3 Software Configuration and Build Management 55
3.3.1 Source Code Control – SCCS/RCS 55
3.3.2 Build Management 55
3.3.2.1 Make 55
3.3.2.2 Smart Recompilation 56
3.3.3 Other Configuration Management Tools 56
3.4 Tools Based on Static Program Analysis 57
3.4.1 Compiler Supporters 57
3.4.2 Data Flow Analysis 57
3.4.3 Cross-Referencers 58
3.5 Meta-Databases 58
3.5.1 History of Development 58
3.5.2 Standards 59
3.5.3 Features of Meta-Data Systems 61
3.5.4 Commercially Available Products 62
3.5.4.1 System Catalogues 62
3.5.4.2 Data Dictionaries 62
3.5.4.3 Repositories 62
3.6 Support Environments 63
3.6.1 Language Independent Support Environments 63
3.6.2 Language Specific Support Environments 64

3.6.2.1 APSE ... 64
3.6.2.2 Other Closed Environments ... 64

3.7 Summary ... 65

Chapter 4: Enabling Technology **66**

4.1 Persistent Programming ... 66

4.2 Napier88 ... 68
4.2.1 Types ... 68
4.2.1.1 Type Databases ... 69
4.2.2 Higher-Order Procedures ... 69
4.2.3 Environments ... 70
4.2.3.1 Type Checking and Binding ... 71
4.2.3.2 Separate Compilation ... 72
4.2.3.3 Some Napier88 Programs' Impact on the Persistent Store ... 73

4.3 The Napier88 Programming Environment ... 76
4.3.1 The Maps Library ... 77

4.4 Napier88 Language Processing Technology ... 77

4.5 Summary ... 78

Chapter 5: TSIT – A Thesaurus-Based Software Information Tool **79**

5.1 Introduction ... 79

5.2 The Napier88 Thesaurus ... 80

5.3 Querying the Thesaurus ... 84

5.4 Registration and Update ... 85

5.5 Implementation ... 87

5.6 TSIT versus other Tools ... 88

5.7 Measuring Name and Identifier Usage – A TSIT Experiment ... 90
5.7.1 Scale of Analysis ... 92
5.7.2 Name Frequencies ... 93
5.7.3 Kind ... 95
5.7.4 Name Usage and Context ... 96
5.7.5 Constancy ... 98
5.7.6 Name Length ... 100
5.7.7 Use of Type Definitions ... 101
5.7.7.1 Use of Structure Fields and Variant Tags ... 104
5.7.8 Use of Procedures ... 105
5.7.8.1 Consequences of Change to Procedures ... 106
5.7.8.2 Context of Procedures ... 107
5.7.8.3 Polymorphic and Specialised Procedures ... 108
5.7.9 Measurements Related to Environments ... 110
5.7.9.1 Changes to Environments ... 111

5.8 Summary 113

Chapter 6: Models and Methodologies 115

6.1 Introduction 115
6.1.1 Motivation 116
6.1.2 Requirements for Models and Methodologies 117

6.2 A Structured Persistent Application System Model – SPASM 118
6.2.1 A Persistent Location Binding Methodology 119
6.2.2 Program Categories 120
6.2.3 Binding Categories 121

6.3 The SPASM Constraints 122
6.3.1 Program Categories 125
6.3.2 Type Definitions 125
6.3.3 Declaration and Use 127
6.3.4 Stub Constraints 129
6.3.5 Drop-Clauses 131
6.3.6 Order of Insert-Programs and Type-Programs 131
6.3.7 Structuring and Naming Conventions 133
6.3.8 Persistent Store 135

6.4 Actions to Conform to the SPASM Constraints 136

6.5 Future Development of a Maintenance Methodology 138
6.5.1 Modifying Procedure Types 139
6.5.2 Modifying Directories and Environments 142
6.5.3 Modifying Types – Schema Evolution 143

6.6 Summary 145

Chapter 7: EnvMake – A Persistent Programming Tool 147

7.1 Introduction 147

7.2 Information about Application Structure 148

7.3 Supporting the SPASM Model 151
7.3.1 Checking the SPASM Constraints 152
7.3.2 Flexibility of EnvMake 154
7.3.3 User Experiences 154

7.4 Build Management 155
7.4.1 Showing Status Information 156
7.4.2 Compilation 158
7.4.3 Execution 160
7.4.4 Installation 160

7.5 Implementation 161
7.5.1 Problems with Ensuring Up-To-Date Information 161
7.5.2 Problems of Naming and Identity 162
7.5.2.1 Returned Environments 163

7.5.2.2 Environments in other Data Structures 164
7.5.2.3 Aliases to Environments 165

7.6 Future Development of EnvMake 165

7.7 Summary 167

Chapter 8: Conclusions and Future Work **169**

8.1 Summary – Utilisation of Thesauri 169
8.1.1 Quantifying Evolution 170
8.1.2 Thesauri in a Strongly Typed Persistent Environment 171
8.1.3 Models and Methodologies 173
8.1.4 EnvMake 174
8.1.4.1 Structure and Dependency Visualisation 174
8.1.4.2 Supporting Steps of the Construction and Maintenance Methodology 174
8.1.4.3 Checking the SPASM Constraints 175
8.1.4.4 Build Management 175

8.2 Future Work – Further Utilisation of Thesauri 176
8.2.1 Schema Evolution 178
8.2.2 Persistent Software Configuration Management 179
8.2.3 Extensibility of SPASM 180
8.2.4 Automatic Generation of Use-Clauses 180
8.2.5 Referencing Environments 181
8.2.6 Hyper-Programming 182
8.2.7 Further Measurements 184

8.3 Finally 185

Appendix A: HMS Execution Log **187**

Appendix B: TSIT Measurements **189**

Bibliography **194**

Index **209**

List of Figures

Figure 1.1: Building and maintaining application systems 8
Figure 2.1: The main components of the HMS system 15
Figure 2.2: The Thesaurus relation 17
Figure 2.3: Definitions and uses of names distributed by name_type 19
Figure 2.4: The Query_Dictionary relation 20
Figure 2.5: The Versions_Thesaurus relation 20
Figure 2.6: The thesaurus interface 21
Figure 2.7: The thesaurus scripts and programs 27
Figure 2.8: Change history of the relations 35
Figure 2.9: Change history of the fields 35
Figure 2.10: Direct and indirect use of relations and fields 36
Figure 2.11: Consequences of the December 1991 HMS schema modification 38
Figure 2.12: Extension of the system structure 42
Figure 3.1: The software development and maintenance process 46
Figure 3.2: Concepts in software construction and maintenance 47
Figure 3.3: The IRDS levels and pairs 60
Figure 4.1: The three mappings of a traditional database system 67
Figure 4.2: The only mapping of a persistent system 67
Figure 4.3: Operations on environments 70
Figure 4.4: Part of the store after running Prog1.N and Prog2.N 73
Figure 4.5: Part of the store after running Prog3.N 74
Figure 4.6: Part of the store after running Prog4.N 75
Figure 4.7: Part of the store after running Prog4.N and Prog2.N 75
Figure 4.8: Part of the store after running Prog6.N 76
Figure 5.1: Definition of thesaurus entry 82
Figure 5.2: The program writePerson.N 83
Figure 5.3: Thesaurus definition 88
Figure 5.4: Name frequency 94
Figure 5.5: Name usage – total 96
Figure 5.6: Name usage – by application 97
Figure 5.7: Proportion of constants in the applications 99
Figure 5.8: A vector program 100
Figure 5.9: Distribution of name length 101
Figure 5.10: Distribution of use of type definitions 102

Figure 5.11: Distribution of use of procedures 106
Figure 6.1: Relationship between SPASM and the methodology 116
Figure 6.2: Binding categories 122
Figure 6.3: ER diagram of programs, bindings and type definitions 124
Figure 6.4: A partial order in the set of programs 132
Figure 6.5: Linear sequence after topological sorting 132
Figure 6.6: Environment structure in persistent store 134
Figure 6.7: Strategy 1 140
Figure 6.8: Strategy 2 141
Figure 6.9: Strategy 3 142
Figure 7.1: The EnvMake menu 149
Figure 7.2: Type dependencies 159
Figure 7.3: Environment as result type 163
Figure 8.1: Thesaurus-based tools 172
Figure 8.2: More thesaurus-based tools 177
Figure 8.3: Methodologies and tools as input to a new language 186

List of Tables

Table 2.1: NAME_TYPE distributed by CONTAINER_TYPE and DEFINITION_USE 18
Table 2.2: Excerpt from the Versions_Thesaurus relation 24
Table 2.3: Added and deleted relations and fields in the HMS schema 34
Table 2.4: Direct use of relations and fields in the query dictionary 37
Table 2.5: Indirect use of fields in Display Language and Hippo code 37
Table 2.6: Consequences of the December 1991 HMS schema modification 39
Table 3.1: Categories of support environments 63
Table 5.1: The corresponding thesaurus entries for the program writePerson.N 83
Table 5.2: The HMS thesaurus tool versus TSIT 89
Table 5.3: Lines of code 93
Table 5.4: Name occurrences 93
Table 5.5: Name use within programs 94
Table 5.6: Number of times a name is declared within a program (percentages) 95
Table 5.7: Distribution of kind 96
Table 5.8: Name usage by application 97
Table 5.9: Distribution of context 98
Table 5.10: Constancy distributed by usage 99
Table 5.11: Corresponding thesaurus entries 100
Table 5.12: Name length of type and value identifiers 101
Table 5.13: Statistics on the use of type definitions 102
Table 5.14: Use of structure fields 104
Table 5.15: Use of variant tags 104
Table 5.16: Kind of structure fields 105
Table 5.17: Kind of variant tags 105
Table 5.18: Use of procedures 107
Table 5.19: Context of procedures 107
Table 5.20: Use frequency and number of types instantiated 109
Table 5.21: Specialised procedures 109
Table 5.22: Usage and context of specialised procedures 110
Table 5.23: Number of name occurrences related to operations on environments 111
Table 5.24: Programs modifying environments 111
Table 5.25: Environments modified by a program 112
Table 5.26: Programs modifying an environment 113

Table 6.1: The SPASM constraints 123
Table 6.2: Unused type definitions 127
Table 6.3: Relationship between type identifiers and type names 127
Table 6.4: Unused value identifiers 129
Table 6.5: Update of procedure variables 130
Table 6.6: File naming conventions 134
Table 6.7: Actions to reconform to constraints that have been violated (continues) 136
Table 6.7: Actions to reconform to constraints that have been violated (continued) 137
Table 6.8: Impact of adding, renaming or deleting a type definition 144
Table 7.1: Parameters of the envMake command 148
Table 7.2: Insert-update dependency table 148
Table 7.3: Use-stored dependency table 149
Table 7.4: Excerpt from a program-environment matrix 150
Table 7.5: Insert-update dependency table 153
Table 7.6: Compilation and execution plan 157
Table 7.7: Log of compilations and executions 157
Table 7.8: Declaration of environments 164
Table 8.1: Platform improvements 186
Table B.1: Frequencies of Kind by Application 189
Table B.2: Frequencies of Context by Application 190
Table B.3: Use of type definitions in value instantiations 191
Table B.4: Environments accessed per program 192
Table B.5: Programs per environment 193

Chapter 1

Introduction

The research presented in this thesis establishes thesauri as a viable foundation for models, methodologies and tools for change management. Most of the research has been undertaken in a persistent programming environment. Persistent language technology has enabled the construction of sophisticated and well-integrated change management tools; tools and applications reside in the same store. At the same time, the research has enhanced persistent programming environments with models, methodologies and tools that are crucial to the exploitation of persistent programming in construction and maintenance of long-lived, data-intensive application systems.

The dominant activity of the large-scale software industry is the production of changes to application systems. Figures describing the maintenance proportion of the total lifetime expenditure on a software system vary between 50% and 90% [Zelkowitz 1978, Lehman 1981, Putnam 1982, Parikh and Zvegintsov 1983, Chikofsky and Cross 1990]. It has been reported that the maintenance proportion was 35-40% in 1970, 40-60% in 1980 and estimated to be 70-80% in 1990 [Pfleeger 1987]. As application systems live longer and grow in size and complexity, it is likely that this trend will continue. The maintenance activities have been divided into the following categories [Swanson 1976]:

i) *Corrective maintenance* (detecting and correcting errors - routine debugging)

ii) *Adaptive maintenance* (accommodation of changes to the environment - including hardware and system software)

iii) *Perfective maintenance* (user requested enhancements, improved documentation, enhanced performance)

It has been reported that the respective categories count for 17%, 18% and 60% of the total maintenance activities (4% in other categories) [Lientz *et al.* 1978]. Within the third category, two thirds were user requested enhancements. This shows that the majority of changes are not due to errors or other causes that one might believe could be prevented by better requirements analysis, design and implementation techniques.

One area of system evolution that has been of particular interest recently is changes to database schemata (schema evolution) [Banerjee *et al.* 1987, Skarra and Zdonik 1987, Lerner and Habermann 1990]. Such changes may have serious impact on other parts of the schema, on extensional data (database objects), on application programs and on user-interfaces. Versioning, build management (compilation, linking and execution) and software configuration management are other major areas relevant to system evolution.

One might argue that the software changes could be reduced by more use of prototyping techniques. Prototyping may enable end-users to express their needs and requirements more accurately in areas such as screen design and certain aspects of system behaviour. However, since new requirements, changing environments, bug-fixing, etc. are encountered after the system has become operational, it is the operational system itself which has to be changed. The challenge is thus to build large, long-lived, data-intensive application systems that can be incrementally modified in compliance with changing user needs. So, reducing the extent of perfective change is not necessarily desirable. It is usual for people carrying out tasks to recognise improved methods and opportunities. Application systems are therefore most likely to support people well if they facilitate change, and allocating resources to at least perfective change should be regarded as valuable.

The approaches to improving the quality, including maintainability, of software application systems may be divided into the following categories:

i) new and improved programming languages (Ada, object-oriented programming languages, persistent programming languages, etc.);

ii) programming guidelines or design principles (e.g. structured programming [Dahl *et al.* 1972, Jackson 1975] or modularisation [Parnas 1972] where a high degree of cohesion[1] and a low degree of coupling[2] among software components should be pursued [Constantine and Yourdon 1979]); and

iii) comprehensive programming methodologies and supporting tools.

The research presented in this thesis focuses on the last approach. The research establishes thesauri as a viable foundation for programming methodologies and supporting tools that are tailored to manage the problem of change.

1 Cohesion is a measure of the degree to which parts of a program module are closely functionally related.

2 Coupling is a measure of the strength of interconnections between modules of a program.

1.1 The Problem of Change

In spite of the proportion of maintenance costs presented above, there is a common misconception that change is something unusual which could be dealt with in an *ad hoc* way. The assumption of stability prevails in current teaching and practice concerning programming, data modelling, database schema construction, etc.

Based on long experience and quantitative studies of several systems, mostly operating systems, Lehman has proposed five "laws" concerning software evolution [Lehman and Belady 1985]. The first two follow:

i) A program must continuously undergo change in order to reflect change in its environment. If not, the program will become less and less useful.

ii) As a large program is continuously changed, its complexity increases, which reflects deteriorating structure, unless work is done to maintain or reduce it.

As shown in [Lientz *et al.* 1978], most changes are due to enhancements in functionality, rather than to bad design, bugs, etc. People do not know in advance or are not able to accurately express all the desired functionality of a large application system. Only experience from using the system in an operational environment will enable the needs and requirements to be properly formulated. The requirements assessment will continuously change during maintenance, and new requirements may be as demanding as those that directed the initial construction.

Many factors may influence change in user requirements: change in market, workforce, skills, economy, legislation, etc. However, this thesis will not discuss the *causes* of change any further, nor will it discuss maintenance problems such as programmer-user communication, programmer effectiveness, etc. [Lientz and Swanson 1981] or other change problems related to project management [Ferraby 1991].

The problem of change is closely related to scale. A whole class of problems only show up when a system becomes long-lived (typically involving persistent data) and grows in size, complexity and diversity (variability). *Persistent application system* or *PAS* will be used throughout this thesis as a succinct phrase denoting large-scale, long-lived and data-intensive application systems that satisfy a complete area of information-processing requirements, for example, a management information system, a health management system or a CAD/CAM system. DeRemer and Kron distinguish "programming-in-the-large" from "programming-in-the-small" and claim that different languages should be used for the two activities. They propose a "module interconnection language" as one necessity for supporting "programming-in-the-large" [DeRemer and Kron 1976]. Further ideas for "programming-in-the-large" (or "mega-programming") are outlined in [Wiederhold *et al.* 1992]. Not only languages, but methodologies and tools are also the subject of new and changed requirements in order to cope with increase in scale. For example, they must support *incremental* design, construction and commissioning.

1.2 Software Maintenance

The term *software maintenance* denotes all changes to the software of an application system after its first installation in its operational environment. Since software systems do not physically wear out or break,[1] software maintenance differs from general maintenance in that the former is not concerned with rectification to an earlier state. Software does not change on its own. It is only changed by people (or possibly by other software such as tools) to adapt to changed requirements, to improve performance or to correct errors.

It is deceptively easy to change software (simple editing), and software is therefore changed much more frequently than tangible products. However, it is not easy to make *consistent* changes; it is easy to cause a mutation but very hard to generate a viable one, particularly if multiple copies have been shipped, etc. A change in one place may have unintended effects elsewhere; even minor local changes can have global impact. Included in the consequences are new errors (the ripple effect). One study found that more than 50% of all errors were due to previous changes [Collofello and Buck 1987].

A challenge is to ensure that *all* consequential changes are dealt with correctly by propagation throughout the system and that no unnecessary changes occur, perturbing working practices and operational software. Long-lived application systems should therefore be designed with change in mind. Moreover, since "the only constancy is change itself", also the organisation should be planned for change [Brooks 1975]. It may be difficult to persuade software builders and managers to plan for change since it requires some extra effort during initial construction which may hinder meeting short-term budget and time goals. The short-term thinking discourages designing for maintenance even though it is an investment that will more than pay off in the long run.[2] Lehman formulates it this way [Lehman 1980]:

> It will, in fact, be suggested that the need for continuing change is *intrinsic* to the nature of computer usage. Thus the question raised by the high cost of maintenance is not exclusively how to control and reduce the cost by avoiding errors or by detecting them earlier in the development and usage cycle. The *unit cost of change* must initially be made as low as possible, and its growth, as the system ages, minimised. Programs must be made more *alterable*, and the alterability *maintained* throughout their life-time.

During maintenance work, a significant part of the time is spent on understanding the existing system. The quality of the software documentation is therefore crucial. However, most documentation is notoriously poor and virtually always obsolete. The only reliable,

1 A physical copy on any medium may deteriorate.

2 This is also a problem regarding software reuse [Krueger 1992], for example.

up-to-date program information may be the source code itself or information that is automatically generated from the source code. The next section therefore introduces the idea of a recording tool that automatically generates and maintains the information that may be required for change and consequential change propagation.

1.3 Thesauri as Foundation for Change Management Tools

Thesauri containing information about names and their usage in all the software of the actual application system (possibly including database schemata) will be proposed as a platform for methodologies and tools that help to solve the problems described in the previous sections. The term *thesaurus* generally denotes "a 'treasury' or 'storehouse' of knowledge, as a dictionary, encyclopædia, or the like" [Oxford 1961]. In the context of this thesis, the "knowledge" is information about names and identifiers such as where they are defined and used, what kinds they are, in which contexts they occur, etc. The term is not used in the more popular meaning denoting a dictionary of synonyms. Alternative terms include the commonly used *data dictionary* and *repository*, but they have been avoided due to confusing terminology in the area and to emphasise the distinction between the thesaurus tools to be introduced and most commercially available data dictionary (repository) tools.[1] The thesauri dealt with in this thesis contain name and identifier information in *all* the software written in *all* the languages of a PAS. This includes source code information about programs in a file system and meta-data information about extensional data in a database or persistent store. This is in contrast to most commercially available tools which focus either on the source code only (source code analysers) or on database-specific information like database schemata (data dictionaries). A few data dictionary tools also include source code information, but relationships between names and identifiers in the software written in the various languages are not recorded automatically.

Names are the focus of attention in this thesis since they are central to system builders' thinking and thus influence the way software is organised. Meaningful names are important for problem solving, understanding of semantic structure and retention – particularly in large and complex application systems [Barnard *et al.* 1982, Weiser and Shneiderman 1987]. The meanings attached to names are relatively stable when dealing with concepts at an abstract level (even though the detailed semantics and interpretation may vary between people and between contexts). This contrasts with all changes in physical software implementations. Therefore, there is potential for tools that help manage the evolution while preserving the use of names.

1 Yet another term, *concordance*, also denotes a collection of name information but generally has a more narrow interpretation than *thesaurus*: "an alphabetical arrangement of the principal words contained in a book, with citations of the passages in which they occur" [Oxford 1961].

It should be emphasised that from a language processing point of view, names are uninteresting; names are not kept by the system. Even names denoting types are irrelevant to language processors. However, names are added in order to make types and other parts of software meaningful to humans. Appropriate tools in the programming environment should support people in the naming process.

Names of identifiers used in an application system may refer to real-world objects or classes of objects modelled by the system (e.g. *person*), to objects specific to the implementation (e.g. *index*) or simply to nothing but the associated location in a memory (e.g. a counting variable *i*). A focus on names may encourage people to be more conscious of what the names are supposed to refer to (though the semantic relation between names and what they refer to is a classical, largely unresolved problem [Nelson 1992]). Choosing names carefully would also prevent name ambiguity.

The vision of this research is that automatically generated and updated thesauri, in addition to serving as a software information repository for application systems builders, can serve as a viable foundation for change management or maintenance tools. The following sections describe the work that has been undertaken in order to test and realise that vision. Two thesaurus tools have been built. The HMS thesaurus tool was developed for a health management system (HMS) in an industrial (C, C^{++}, X-windows and relational database) environment [Sjøberg 1991]. Another thesaurus tool was thereafter built in the context of the strongly typed, persistent programming language Napier88.

1.4 A Thesaurus Tool in an Industrial Environment

The HMS thesaurus tool was developed in an industrial environment in order to identify and help solve real-world problems of maintenance. All the software is analysed by the thesaurus tool. This includes programs written in a screen definition language, a procedural language for defining actions, a query dictionary language and a schema definition language. The tool stores information about name occurrences, like type and container, and records dependencies between occurrences in all the software (including between software written in different languages). An interface provides, among other things, some consistency checking and impact analysis which localises the effects of change within the system. The impact analysis of schema changes has proved particularly useful since software written in all the languages is affected by such changes; doing the same analysis manually is a very hard task.

1.4.1 Schema Evolution Measurements

Some measurements of system evolution have been reported in the literature [Lientz *et al.* 1978, Lehman and Belady 1985], but in order to turn computing science into an exact science,[1] more measurements should be provided. Regarding change management, measurements are a useful supplement to anecdotal information when, for example, performing the following tasks:

i) identifying form and extent of various kinds of change;

ii) designing change management methodologies and tools; and

iii) testing the usefulness of such methodologies and tools.

In order to acquire a deeper understanding of the nature of system evolution, measurements of change in the HMS system were collected over a period of 18 months, both during initial construction and after the system became operational. The thesaurus tool was enhanced to monitor the evolution, which was studied in general but changes to the schemata were emphasised. The extent of schema evolution (e.g. 140% increase in number of relations) and its consequences confirms the need for supporting tools [Sjøberg 1993]. The method for, and the results of, these change measurements are one step further in the direction of quantifying change. The cost and effort in conveying in-depth analyses of real-life, industrial projects may be one reason for the lack of references to other schema evolution studies in the literature.

1.5 Thesaurus-Based Tools in Persistent Environments

The basic problems of change are the same whether the environment is an industrial relational database context with conventional programming languages or an experimental object-oriented database context with more modern programming languages, etc. The experiences with the HMS thesaurus tool therefore served as a general basis for further research in change management. The built-in provision for longevity should enable persistent programming technology to be a suitable platform for such research.

The major motivation for persistent programming language research is the belief that such languages will facilitate the construction and maintenance of PASs [Atkinson *et al.* 1982, Atkinson *et al.* 1983a]. Persistence was invented more than a decade ago, and persistent languages such as Napier88 [Morrison *et al.* 1989a] have proved robust and well engineered [Sjøberg *et al.* 1993]. However, to fully benefit from the potential advantages of persistence in the construction and maintenance of large PASs, suitable programming methodologies and supporting tools have to be developed. An ultimate goal would be to integrate all needed tools in a persistent software engineering environment.

1 One can of course question to what extent it is possible to make a science of artificial (i.e., human-made) systems exact [Lehman 1976].

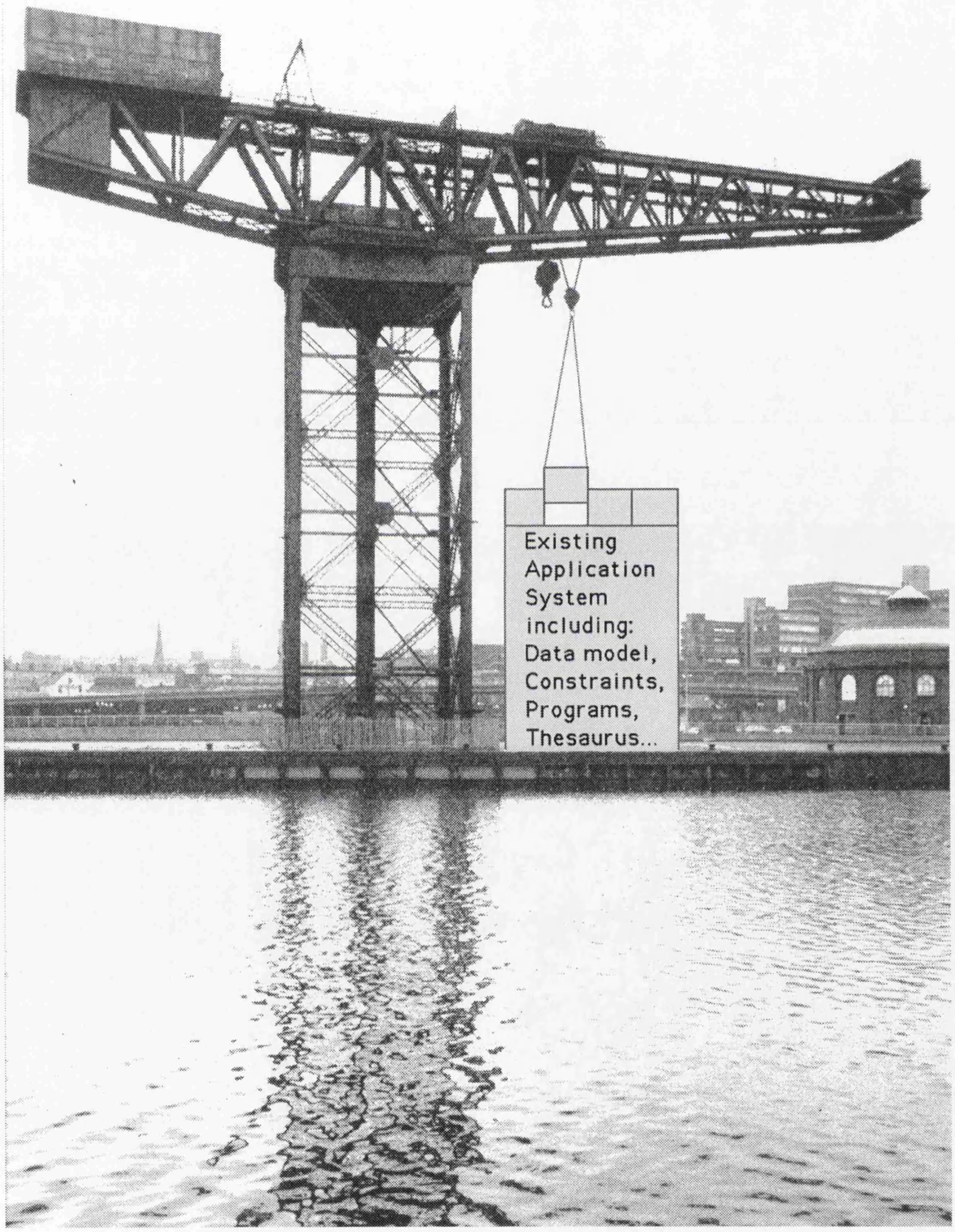


Figure 1.1: Building and maintaining application systems

There were two reasons for choosing Napier88 as the experimentation language. First, the features of the language indicated it as suitable for research in change management. Second, the methodologies and tools to be developed would enhance the programming environment of Napier88 itself. Napier88 was (and still is) in its infancy as an implementation language for large-scale applications. For example, guidelines and tools are still needed to organise the interaction between programs and bindings in the persistent store.

The model for constructing and maintaining persistent application systems pursued in this thesis may be illustrated by Figure 1.1. Powerful tools generate application system increments from a repository of data models, constraints, programs and thesauri. The tools to be described in this research include cross-referencers, impact analysers, constraint checking tools, build management tools, etc. Similar tools have been built in other programming environments, but the research includes several aspects specific to persistent programming and is also original in other respects (e.g. automatic, incremental collection of information).

1.5.1 A Persistent Thesaurus Tool

The thesaurus tool built in and for Napier88 collects fine-grained information about names in the source programs and names denoting bindings in the persistent store. Compared with the HMS case, it is in one sense simpler to extract the thesaurus information for Napier88 since only one language needs to be analysed. On the other hand, the kind of information stored in the thesaurus for Napier88 is more complex in order to capture the additional information that Napier88 provides, as it is a sophisticated language with a rich, strong type system including environments, polymorphic procedures, abstract data types, etc.

A textual interface to the thesaurus was built by the author; a comprehension query language (enabling recursive queries) and a window-based interface to the thesaurus were built by others [Sjøberg *et al.* 1993]. Integrating these interface components was easy due to the persistent technology.

Eight Napier88 applications were measured in a study conducted with the intention of identifying how persistent programmers use the constructs of Napier88 and how they organise their software. The results may be useful for optimisation of language implementation and for design of methodologies and tools. Measurements of source programs have been collected for other languages such as FORTRAN [Knuth 1972], PL/1 [Elshoff 1976], APL [Saal and Weiss 1977] and Ada [Agresti and Evanco 1992], but the measurements to be presented are the first in the context of a persistent programming language.

1.5.2 Models and Methodologies

The term *methodology* may be interpreted as "the processes, techniques, or approaches employed in the solution of a problem or in doing something: a particular procedure or set of procedures" [Webster 1961].[1] Within the scope of this thesis the "problem" or the "something" being done is *construction and maintenance of PASs*. The terms *construction methodology* and *maintenance methodology* will thus respectively be used to denote a model for the initial construction of PASs and a model for the maintenance of such systems. The methodologies should constrain software builders' software with the purpose of developing intelligible, correct and efficient application systems that remain relatively easy to change.

Some components of an overall methodology for persistent programming have already emerged, for example an architecture for organising an application around persistent L-value procedures which can be incrementally modified without the need for editing, recompiling or re-executing the programs containing the callees [Cutts 1993a]. Nevertheless, other guidelines and more complete methodologies are still needed for other aspects of persistent programming. This thesis contributes new aspects of a construction and maintenance methodology. This includes the SPASM model which is a set of constraints to which each suite of application software should adhere in order to ensure correctness and maintainability. Several of the constraints are based on a categorisation of programs into five groups according to their semantics. This categorisation, which is done automatically, is also the basis for build management (see next section).

Both SPASM and the other components of the methodologies are general in that they are independent of the actual real-world applications being implemented. They are, however, couched in terms of the programming language (Napier88) even though most of the principles they encode are applicable to any database programming environment.

1.5.3 EnvMake – Another Thesaurus-Based Supporting Tool

At present, most Napier88 programmers use Make [Feldman 1979] to help rebuild the application after change. The programmers have to manually specify compilation and execution dependencies. Similarly, Make and sometimes Unix™ shell scripts are used to install software into the persistent store. Nevertheless, a correct installation-order has to be determined and typed in manually into a Makefile or a script. These problems are addressed by EnvMake – a tool that automatically infers the necessary dependencies from the thesaurus and initiates (re)compilation and (re-)execution. If installation is requested, EnvMake installs components in correct order (if such an order exists)[2] into the persistent

[1] This differs from what may be the original meaning: "a science or the study of method" [Webster 1961].

[2] If components cyclically refer to each other, installation is impossible.

store. EnvMake thus relieves the programmers from the burden of maintaining Makefiles and scripts.

In addition to replacing the use of Make, EnvMake provides additional functionality tailored for persistent application development such as checking the SPASM constraints and presenting information about which programs carry out which operations on which bindings in the persistent store.

1.6 Thesis Statement

Measurements show that application programming deals with a very high rate of change to data definitions, dependent programs and dependent user interfaces. This leads to severe problems propagating changes correctly. It is common to find that necessary changes consequent on some other change have not been made, so that the system is inconsistent and will *eventually* fail to operate correctly.

Methodologies and tools based on thesauri, containing automatically generated information about programs and data, have been proposed, prototyped and demonstrated. It is argued that investing in following methodologies and using supporting tools would achieve significant improvement in application programmer productivity, including the ability to manage change. It is further demonstrated that such methodologies and tools are pertinent to strongly typed persistent systems and can be integrated into an effective persistent programming environment.

1.7 Thesis Structure

Chapter 2 describes an industrial experiment conducted with the intention of identifying problems related to change management and building tools to help solve the problems. Measurements of the extent of change and change consequences are provided.

Chapter 3 surveys diverse techniques and tools in software engineering with emphasis on maintenance.

Chapter 4 presents persistent language technology as a platform for the research described in the subsequent chapters.

Chapter 5 describes how the ideas behind the industrial experiment (Chapter 2) are further developed in a thesaurus tool built in the context of the strongly typed, persistent programming language Napier88. The usefulness of the tool is, amongst others, demonstrated in an explorative study providing a plethora of measurements useful for language designers, tool builders and application programmers.

Chapter 6 introduces models and methodologies for persistent application construction and maintenance. As a means to improve correctness and maintainability, the SPASM model defines a set of constraints to which PASs should adhere. Measurements of eight Napier88 applications describe to what degree existing software complies with those constraints. The chapter also discusses steps of a construction methodology and actions of a maintenance methodology that should be carried out, depending on the kind of change.

Chapter 7 presents a tool, EnvMake, which checks the constraints of the SPASM model and supports the steps of the construction and maintenance methodology. It also describes how EnvMake features automatic installation, (re)compilation and (re-)execution, and how it visualises dependencies between identifiers in source programs and bindings in the persistent store in the form of dependency tables or matrices.

Chapter 8 concludes by emphasising thesauri as a foundation for change management methodologies and tools. The chapter summarises the achievements of the thesis and outlines further work also based on thesauri.

Chapter 2

The HMS Thesaurus Tool – An Industrial Experiment

2.1 Introduction

In order to relate the research issues described in this thesis to the "real world", a large, industrial database application – a health management system (HMS) – was investigated in detail. The purpose was twofold: first, to design and implement a tool that should assist in the current development and maintenance by providing information about the structure of the system; second, to monitor the evolution of the system and collect change measurements.

When the investigation started, the scale and complexity of the HMS system complete with applications indicated that aids to keeping track of the structure of the system were becoming essential. The *thesaurus tool* was developed to become such an aid. Its main component is the thesaurus – a meta-database containing information about the names and their usage in all the software (including the database schemata[1]) constituting the HMS system. On the basis of the thesaurus, the tool helps answer questions such as:

- Which actions, classes, functions, macros, etc. are defined and where are they used?
- Which fields and relations does this query or update function refer to?
- Which actions are referenced in this Display Language program?
- Does this name already denote an action?

[1] There are several schemata in the HMS system – one for each subsystem, e.g. BED BUREAU and GP.

The thesaurus tool also provides impact analysis, consistency checks and change history information. Another feature of the tool is that it may serve as a measurement apparatus. This was demonstrated in an experiment undertaken with the purpose of acquiring a deeper understanding of the nature of evolution [Sjøberg 1993]. A particular goal was to quantify the problem of changes to database schemata and necessary change propagation. The HMS system was observed over a period of 18 months. The study to be reported illustrates how significantly the schema changed and furthermore that even a small change to the schema may have major consequences for the rest of the application code. The measurements confirm the need for methodologies and tools for managing the consequences of changes to database schemata. The measurements also help identify the requirements of such methodologies and techniques.

2.1.1 The HMS System

Relational database management systems (RDBMSs) are currently in widespread use in industry and commerce. The HMS system is one example of an application system utilising this technology. The system, running on high resolution colour Unix™ workstations, consists of *Display Language* and *Hippo* programs [Clifton 1990, England and Selwyn 1990],[1] a *query dictionary* and a database including the associated *schema* (Figure 2.1).

Applications are written as a graph of screens so that a user works via the icons and fields on screens and navigates to other screens in the graph using "buttons". The screens of the user interface are programmed in the Display Language. A Display Language program contains *classes* and *objects* that both represent windows and have attributes that describe properties of these windows. Objects can be defined within classes and within other objects. A class can be used as the type of another class or as the type of an object. It is possible to modify the type of an object by adding attributes or by introducing new objects within the original object in a form of inheritance hierarchy. So, the class-object relationship is not exactly the same as the one found in traditional object-oriented languages. The Display Language is an interpreted language implemented in C and the X Window System.

The procedural part of the user interface is programmed in the Hippo language. An *action* is the main language construct. An action can be global, or it can be local to a *script* which in turn may be associated with a main class in a Display Language program. Hippo is an interpreted language implemented in C.

[1] These languages have recently been integrated into the Polyhedra programming environment [PSL 1992].

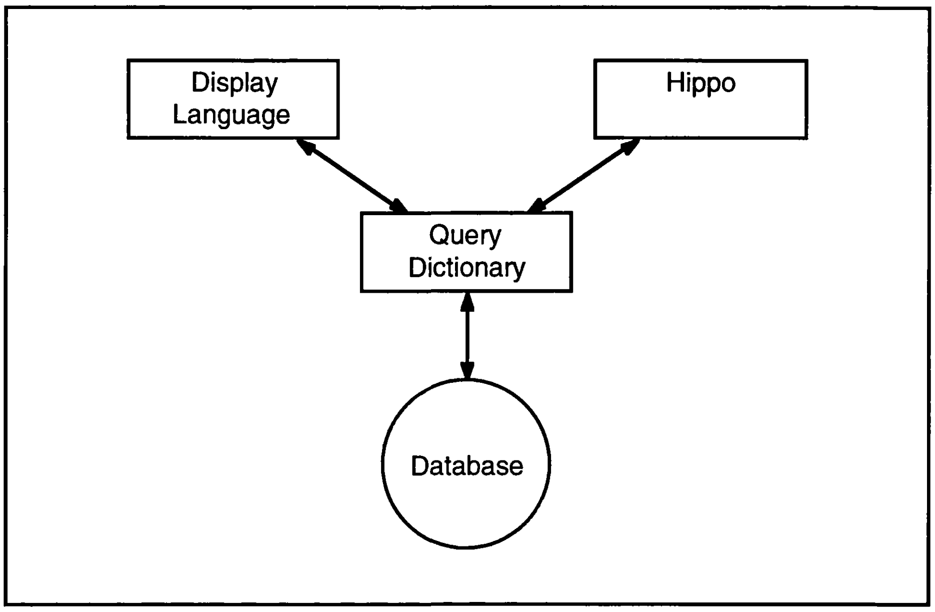


Figure 2.1: The main components of the HMS system

The query dictionary consists of *queries* (SQL *select*) and *update functions* (SQL *insert, update* and *delete*) which are used by the Display Language and Hippo programs when operating on the database. Several update functions may be defined in a *transaction* (usually to ensure referential integrity after update). The query dictionary concept was introduced in the HMS architecture to isolate as far as possible the Display Language and Hippo code from the database. This permits some of the changes to the schema to be hidden from the application code by rewriting the queries and update functions. These queries and update functions are referred to by name with named parameters (Hippo variables) called *datums*.[1] The queries return their results in tables whose columns are also referred to as datums and which may be traversed or automatically displayed. The query dictionary is intended to be sufficiently general not only to absorb change that need not be propagated further, but also to allow different DBMSs to be used and even different data models. The query dictionary is implemented in the Pro*C™ embedded SQL language.

The description of the relations, including their fields, constitutes the *schema*. The actual DBMS is Oracle™.

2.2 The Thesaurus Tool

The thesaurus tool generates the names and associated information by analysing all the software and performs subsequent updates of the thesaurus data. A definite requirement of the thesaurus tool – which has been satisfied – was that the contents of the thesaurus

[1] Plural of *datum* is *data*, but HMS uses *datums* to denote several occurrences of the special HMS concept *datum.*

should not need to be manually maintained. Experience shows that this is crucial for the use of such a tool. The source programs and database schemata are periodically (each night) scanned to detect and record changes. If there is a need for a more up-to-date version, a programmer may also initiate a scan. All the executions are carefully logged.[1] The scan and update are implemented by using a combination of Unix *csh*, *awk* and *sed* scripts, one C program and some Oracle commands for loading and unloading thesaurus data.

An interface consisting of windows with pull-down menus, buttons, etc. is implemented in the Display Language and Hippo themselves and features the following:

- search, sort and display of name information;
- predefined queries for consistency checks like detecting names defined but not used, and names used but not defined;
- finding consequences of proposed change; and
- change history in the form of added and deleted thesaurus entries within a user-specified time interval.

Most of the features of the thesaurus tool are based on cross-reference information also found in other programming environment tools like source code analysers and data dictionary tools [Bourne 1979, IBM 1980, DEC 1989, SoftwareAG 1990], except that the thesaurus tool spans *all* the languages used to build the *whole* persistent application system, its user interfaces and its databases, and thesaurus tool *automatically* collects the data in quiescent periods.

2.2.1 The Meta-Data Relations

Like the HMS databases, the thesaurus is a relational database. It consists of three relations keeping meta-data information. The main one is the *Thesaurus* relation which contains information about the names in the HMS system. The *Query_Dictionary* relation describes the correspondence between the fields of the relations and the datum names used in the Display Language and Hippo programs. In order to keep historical information, with the intention of studying the nature of change, the *Versions_Thesaurus* relation was introduced.

2.2.1.1 The Thesaurus Relation

The seven fields of the Thesaurus relation are described in Figure 2.2. The values of NAME_TYPE correspond to the main constructs of the languages being used. The DEFINITION_USE field indicates whether the container is the place where the name is defined or where it is used. The REMARK field holds a comment on the name. The

[1] Appendix A shows an excerpt from an execution log.

programmer specifies the comment after a special symbol (##) following the definition of a name. Only comments on definitions are supported since it proved difficult to find a handy syntax for giving comments wherever a name can be used.

- SEQ_NO – system-generated key
- NAME – textual form of the identifier
- NAME_TYPE – one of the following codes appropriate to the type of the identifier:
  - AN (Action Name)
  - AS (Action Script name)
  - CN (Class Name)
  - DN (Datum Name)
  - FN (Field Name)
  - FU (FUnction name)
  - QN (Query Name)
  - RN (Relation Name)
  - SM (Screen Macro name)
  - TN (Transaction Name)
  - UN (Update function Name)
- CONTAINER – a textual form of where a name is used
- CONTAINER_TYPE – codes appropriate to the type of the CONTAINER value:[1]
  - AS (Action Script)
  - DL (Display Language program)
  - HP (Hippo Program)
  - QN (Query)
  - QD (Query Dictionary)
  - RN (Relation)
  - SC (Schema)
  - TN (Transaction)
  - UN (Update function)
- DEFINITION_USE (D/U) – indicates definition or use of the name
- REMARK – a comment on the name

Figure 2.2: The Thesaurus relation

Determining the container types for the respective definitions and uses is not straightforward. For example, relations are naturally defined in the schema, but where are they used? At least indirectly, they are used in all kinds of application programs. However, they are only used directly in queries and update functions, which therefore have been chosen as the container types. Similarly, one may think of fields as being defined in the schema. However, since they are always defined as part of a relation, relation has become their container type. This also ensures uniqueness of a (field name, container value) pair. A (field name, "schema") pair would not necessarily have been

[1] The reason for the apparently unnatural choice of CONTAINER_TYPE codes in some cases is that they should match the NAME_TYPE codes where there are correspondences.

unique. The relationships between the NAME_TYPE, CONTAINER_TYPE and DEFINITION_USE fields are illustrated by Table 2.1. The leftmost column contains the various name types. The other columns represent the container types. A 'D' ('U') in a cell indicates that a name of the given name type can be defined (used) in a container of the given container type.

NAME_TYPE	CONTAINER_TYPE								
	AS	DL	HP	QN	QD	RN	SC	TN	UN
AN (Action Name)	D, U	U	D, U						
AS (Action Script name)			D, U						
CN (Class Name)		D, U							
DN (Datum Name)	U	U	U	D					D
FN (Field Name)				U		D			U
FU (FUnction name)	U		D, U						
QN (Query Name)	U	U	U		D				
RN (Relation Name)				U			D		U
SM (Screen Macro name)		U							
TN (Transaction Name)	U		U	D					
UN (Update function Name)	U		U		D			D	

Table 2.1: NAME_TYPE distributed by CONTAINER_TYPE and DEFINITION_USE

By November 1991, the HMS system comprised about 150,000 lines of source code, but the thesaurus provides a better measurement of the size: the number of programmer-introduced names of various types. Figure 2.3 shows the proportion of definitions and uses for each name type. There are 9152 defined names which are used 15098 times, i.e., a total of 24250 name occurrences. These measurements include only unique occurrences within a container type. That is, if, for example, a datum is referred to several times within an action, it is registered as only one entry in the thesaurus. Information about duplicated name occurrences within a container was not considered necessary for the HMS project.[1] If several entries for the same name were to be recorded, then an indication of the place within the container should also be present (e.g. line number and possibly word number within a file). However, including duplicates would have increased the volume of the thesaurus by 100%.

1 Among the container types, duplicates occur only in action scripts, Display Language programs and Hippo programs.

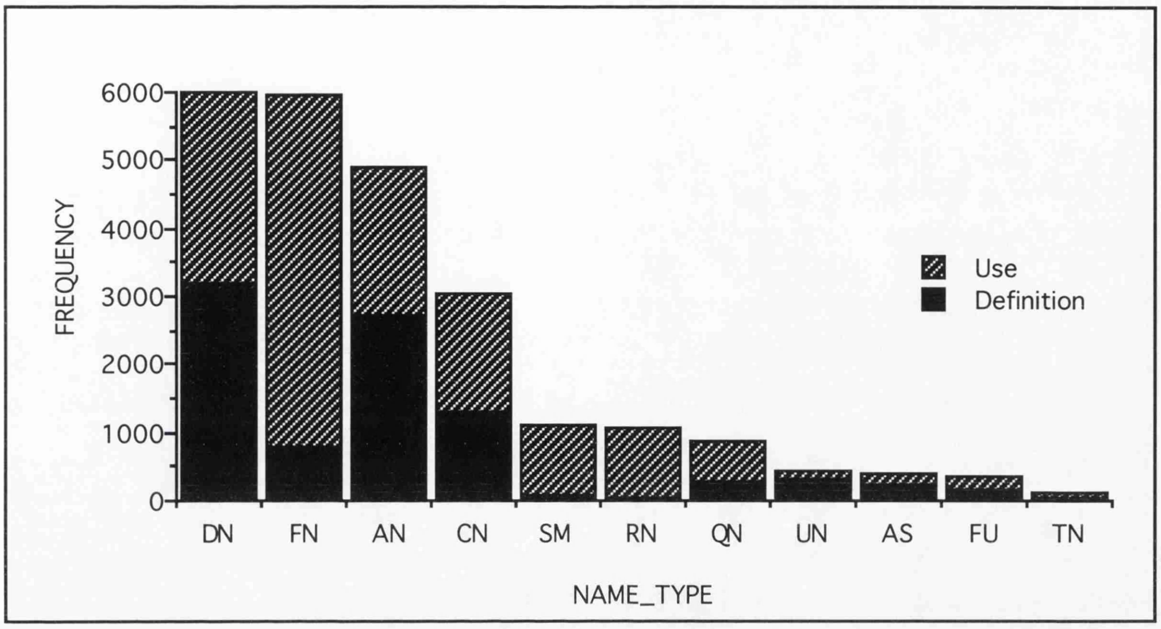


Figure 2.3: Definitions and uses of names distributed by NAME_TYPE

The apparently low use of action scripts and update functions should be explained. There are 168 action scripts that are called in the Hippo code. Another sort of use is that an action script may be associated with a class having the same name as the script. There are 128 such associations. Among the 322 defined update functions, 237 are contained in transactions and are thus only called implicitly when the associated transaction is called.

Normally, we would expect fewer definitions than uses. Figure 2.3 shows, however, that actions (AN), datums (DN), functions (FU) and update functions (UN) all have more definitions than uses. The major reason for this inconsistent state is that the data was recorded in a very active part of the development. At that stage it is natural to define names that are not yet referenced in programs.

2.2.1.2 The Query_Dictionary Relation

In order to find the effects of changes to schemata, queries and update functions, the Query_Dictionary relation was introduced which describes *direct* correspondences between fields of the relations and datums used in the Display Language or Hippo programs (Figure 2.4). This information cannot generally be inferred from the Thesaurus relation since neither a field nor a datum is globally unique. For example, a field SURNAME may be the surname of a patient or the surname of a GP. However, a field is unique within a relation, and a datum is unique within a *QDfunction*[1].

[1] A QDfunction denotes either a query or an update function defined in the query dictionary.

- RELATION
- FIELD
- QDFUNCTION – a name of a query or update function
- DATUM

Figure 2.4: The Query_Dictionary relation

2.2.1.3 The Versions_Thesaurus Relation

The Versions_Thesaurus relation is like the Thesaurus relation but for two added fields that specify whether a name has been added or deleted and the date of the incident (Figure 2.5). A change to the name of a relation, for example, has been registered as one deletion and one addition. It is generally impossible for a tool to distinguish between a rename and a deletion followed by an addition without any user provided information. If the structure of the relation changes as well (fields added, deleted or changed), it is also a semantic problem to decide whether the same relation has been modified or a new one has been created.,ation is registered as one deletion and one addition, whereas a change to the type of a field is not captured in the thesaurus at present.

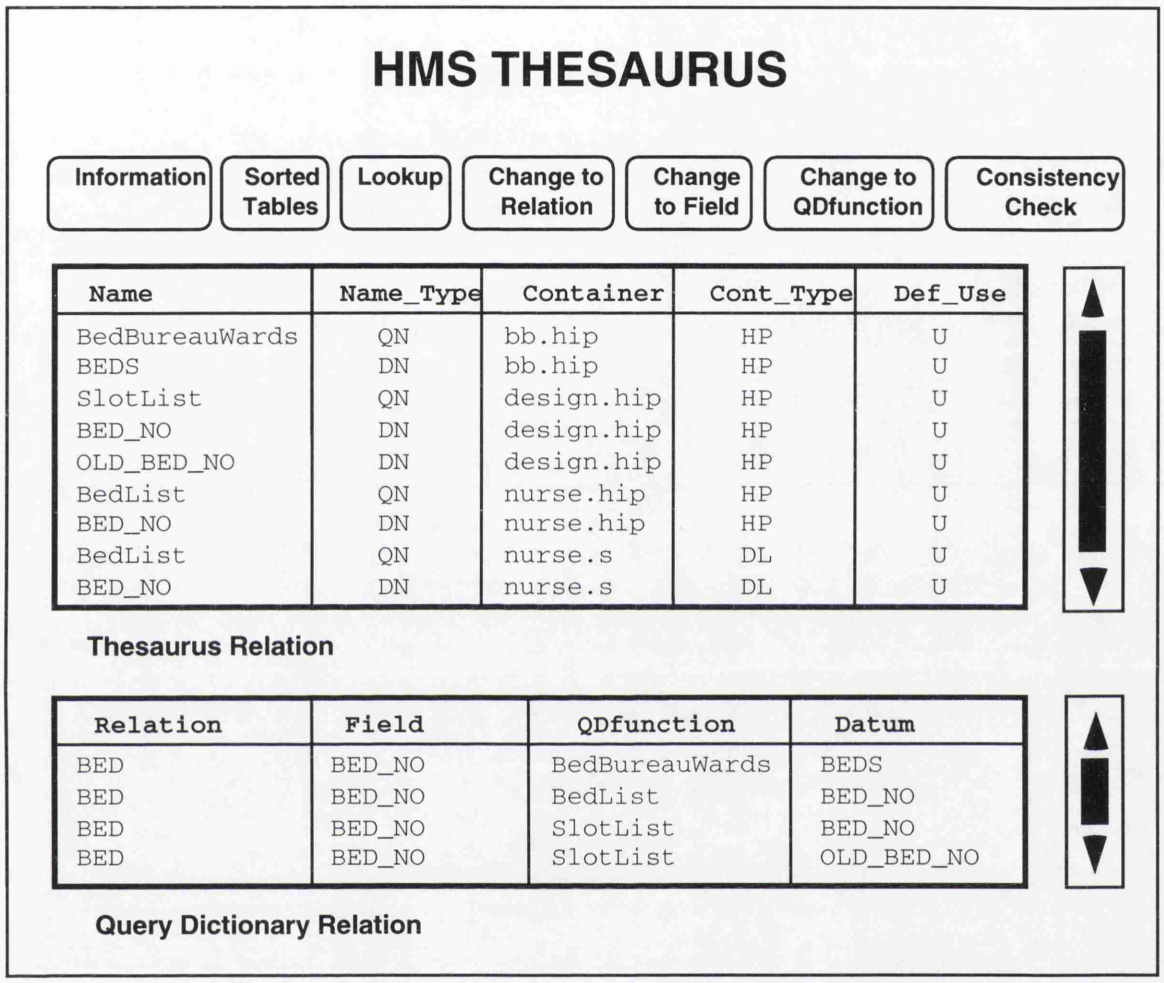
- The fields of the *Thesaurus* relation
- ADD_DELETE (A/D) – specifies whether the name was added or deleted
- INTRODUCED – date of addition/deletion

Figure 2.5: The Versions_Thesaurus relation

An entry with a 'D' in the ADD_DELETE field will always have a corresponding entry with the field value 'A'. The corresponding entry will have a prior INTRODUCED value and incidentally another system-generated key (SEQ_NO value).

2.2.2 The Thesaurus Interface

The interface of the thesaurus tool provides queries for general name information, impact analysis and simple consistency checks. An "*Information*" button informs the user about the structure of the thesaurus relations including the fields and their value sets. The interface has one window displaying information from the Thesaurus relation and another window displaying information from the Query_Dictionary relation. The functionality of the interface is shown in Figure 2.6 which is a sketch of the actual screen (the SEQ_NO and REMARK fields are not included). The real system is implemented using colour-graphics on high resolution workstations. Some (poor quality) screen dumps showing the results of various queries are presented in [Sjøberg 1991].


HMS THESAURUS

Name	Name_Type	Container	Cont_Type	Def_Use
BedBureauWards	QN	bb.hip	HP	U
BEDS	DN	bb.hip	HP	U
SlotList	QN	design.hip	HP	U
BED_NO	DN	design.hip	HP	U
OLD_BED_NO	DN	design.hip	HP	U
BedList	QN	nurse.hip	HP	U
BED_NO	DN	nurse.hip	HP	U
BedList	QN	nurse.s	DL	U
BED_NO	DN	nurse.s	DL	U

Thesaurus Relation

Relation	Field	QDfunction	Datum
BED	BED_NO	BedBureauWards	BEDS
BED	BED_NO	BedList	BED_NO
BED	BED_NO	SlotList	BED_NO
BED	BED_NO	SlotList	OLD_BED_NO

Query Dictionary Relation

Figure 2.6: The thesaurus interface

2.2.2.1 Name Usage Information

To be an efficient and reliable programmer on the HMS project, a certain knowledge of the existing software is essential. In particular, this is a potential problem when a programmer starts working on the project or has been away for some time.

As shown in Figure 2.6, the main screen contains two tables that display data from the Thesaurus and Query_Dictionary relations. Two buttons, "*Sorted Tables*" and "*Lookup*", provide sorted data and lookup of names according to user specified search criteria. Clicking on the "*Sorted Tables*" button invokes a pull-down menu from which the user can select the relation and the fields by which the output should be sorted. Similarly, the pull-down menu of the "*Lookup*" button provides the user with the option of specifying a (substring of a) name and restricting the output by selecting only definitions or uses, a specific name type and/or a specific container type.

As mentioned, even though a name may have several occurrences within a container (a script or program), there is only one entry for the name per container in the Thesaurus relation. It is the responsibility of the programmer to find all the occurrences within the container. It should be a trivial task to use the search functions of an editor or browser to find the exact location of the name occurrences.

2.2.2.2 Schema Evolution – Impact Analysis

As the HMS applications grow, the problem of managing changes to meta-data becomes increasingly difficult. One important category of meta-data is the database schema, i.e., the definitions of the relations and fields. A list of logical changes[1] to a relational schema is:

1) Add a new relation.
2) Rename a relation.
3) Delete a relation.
4) Add a new field to a relation.
5) Rename a field.
6) Change the type of a field.
7) Delete a field from a relation.

Deletion of a relation or field may typically be a consequence of vertically factoring (splitting) relations. Deletion of a relation may also result from joining two relations into one.

The traditional answer to coping with schema changes in a relational database is to interface all software via topic specific views. Changes in the definition of these views are then made to mask the changes in the application programs. However, many of the changes are introduced to effect changes in system behaviour, and these changes must be appropriately propagated rather than masked. Views are not used in HMS since the query dictionary provides the necessary flexibility and indirection. When a change is made to the meta-data, it is necessary to identify which queries and which application programs are potentially affected. The programmer then has to decide where the change should be propagated and where it should be masked.

Impact analysis (also referred to as "what-if analysis") helps in determining where a change should be propagated. If an existing relation or field is to be changed, then an

1 Physical reorganisation is not an issue in this chapter as most RDBMSs absorb such changes – obviating the need to change applications.

impact analysis will inform the places where that relation or field is used. However, there is no indication of where additions should be propagated. It is of course a semantic problem to identify such places, but if a field is added, for example, at least one application program and screen must be changed to collect the new data, and at least one program must eventually use it.

The thesaurus interface provides three "Change to X" buttons (Figure 2.6) which execute queries for finding the name occurrences possibly affected by changes to a relation, field or QDfunction function. Table 2.1 shows that relations (RN) are used in queries (QN) or update functions (UN) which, in turn, are used in Display Language programs (DL) and Hippo programs (HP). Consequently, the query executed by the "*Change to Relation*" button may be explained as: "If this relation is changed, which QDfunctions and thereby which Display Language programs and Hippo programs are then affected?"

The effect of changing a field[1] is more complicated to work out. A field (FN) may be used in several QDfunctions (QN or UN). Since a field is not necessarily globally unique, the relation of the field has to be specified in order to find the affected QDfunctions. Via the QDfunctions, field values are transferred to datums used in Display Language or Hippo programs. Hence, if a field has been changed, the programs containing the corresponding datums will also be affected. The correspondence between a field and a datum, which is not always a one-to-one correspondence,[2] is described by the Query_Dictionary relation.

The "*Change to Field*" button invokes the impact analysis of changing a field. If the query dictionary table of the interface contains some entries (a result of another query), a user can select (say) an occurrence of a field name and then press the button. Figure 2.6 shows an example where the field BED_NO of the BED relation has been selected. In the query dictionary window, all entries having the actual field name are displayed. The thesaurus window displays all occurrences[3] of all datums corresponding to this field and all queries and update functions containing occurrences of the field. The interface works similarly for the other "Change to X" buttons.

Not only schema changes, but changes to other categories of software components may also significantly affect other parts of an application. For example, a change to a QDfunction will affect the Display Language and Hippo programs that use that QDfunction. Therefore, included in the interface is also a "*Change to QDfunction*"

1 The problem discussed concerns changing the *definition* of a field, not its *value*, of course.

2 A datum may obtain its value as a function of several fields; there are 0.62 datums per field on average.

3 In this thesis *occurrence* denotes an occurrence of an identifier – a *name* of a datum, field, etc., not its definition or value.

button which performs a query that finds all programs using a selected query or update function and all relations and fields referred to within that query or update function.

2.2.2.3 Consistency Checks

In a large-scale project such as the HMS project, software will be changed continuously. A high frequency of changes implies a high risk of leaving the application software in an inconsistent state. One kind of inconsistency is that a name denoting an action, class, macro, etc. is defined, but not used. Another, more serious kind, is that a name is used, but not defined. By clicking the "*Consistency Check*" button, the results from the check are displayed on the screen. The results can also be presented in the form of a report.

2.2.2.4 Change History

A second screen of the thesaurus interface displays data from the Versions_Thesaurus relation. A "*Changes Between*" button invokes a query that finds all the thesaurus entries added or deleted within a time interval specified by the user. Table 2.2 shows changes in the interval from 29/6/90 to 12/10/90.

NAME	NAME_ TYPE	CONTAINER	CONT_ TYPE	DEF_ USE	INTRODUCED	ADD_ DELETE
change_ward	AN	diary.hip	HP	D	29-JUN-90	A
change_ward	AN	diary.hip	HP	D	03-AUG-90	D
change_ward	AN	diary.hip	HP	D	10-AUG-90	A
other_login	AN	WardAccess.s	DL	U	20-JUL-90	A
other_login	AN	WardAccess.s	DL	U	03-AUG-90	D
other_login	AN	WardAccess.s	DL	U	07-SEP-90	A
AdminMenu	AS	admin.hip	HP	D	20-JUL-90	A
AdminMenu	AS	admin.hip	HP	D	27-JUL-90	D
AdminMenu	AS	admin.hip	HP	D	17-AUG-90	A
AdminMenu	CN	AdminMenu.s	DL	D	20-JUL-90	A
AdminMenu	CN	AdminMenu.s	DL	D	27-JUL-90	D
AdminMenu	CN	AdminMenu.s	DL	D	17-AUG-90	A
....	..	....	..	.	....	.
....	..	....	..	.	....	.
TIME_OUT	FN	TERMINAL	RN	D	20-JUL-90	A
TIME_OUT	FN	TERMINAL	RN	D	27-JUL-90	D
TIME_OUT	FN	TERMINAL	RN	D	10-AUG-90	A
verify	FU	admin.hip	HP	U	20-JUL-90	A
verify	FU	admin.hip	HP	U	27-JUL-90	D
verify	FU	admin.hip	HP	U	12-OCT-90	A
TERMINAL	RN	SCHEMA	SC	D	20-JUL-90	A
TERMINAL	RN	SCHEMA	SC	D	27-JUL-90	D
TERMINAL	RN	SCHEMA	SC	D	10-AUG-90	A

Table 2.2: Excerpt from the Versions_Thesaurus relation

As a curiosity, the table also shows that some names were introduced and then deleted – and thereafter reintroduced. For example, the relation 'TERMINAL' was introduced 20/7/90, deleted 27/7/90 and then reintroduced 10/8/90. The fields of this relation (e.g. 'TIME_OUT') will of course follow the same course of events. It might be the case that a reintroduced name denotes an object with a different structure and/or semantics than the object denoted by the first introduced name. It is more likely, however, that the "same" object has been reintroduced after – for some reason – having temporarily been removed from the application.

By specifying appropriate clauses on the INTRODUCED and ADD_DELETE fields, one can generate a complete version of the Thesaurus relation that represents the state at any given time. So, in theory the Thesaurus relation is unnecessary since all its information can be obtained from Versions_Thesaurus. Using only the Versions_Thesaurus, however, would be very inconvenient. The implementation of interfaces and queries would be much more complicated, and the performance would be drastically impaired.

2.2.3 Implementation

The generation of the names and the subsequent update of the thesaurus are performed by a combination of C-shell, *awk* and *sed* scripts and one C program. The C-shell scripts use, amongst others, the Unix commands *awk, cpp, diff, grep, sed, sort* and *uniq* [Sun Microsystems 1988a]. The *sqlplus* and *sqlload* commands invoke Oracle. A summary of the use of these commands follows:

- The *awk* macro language facilitates pattern recognition in text files. The lines and words of a file are automatically assigned to (an array of) variables. Other predefined variables contain the file name, number of records, etc. The command *awk* also includes a subset of the C programming language (e.g. the *printf* function which is used to write the names and related information into an appropriate form for the loader). The *awk* scripts constitute the main component concerning the search patterns and rules for finding the names and additional information to be inserted into the Thesaurus relation. The patterns and the rules depend on the name type, container type and whether the definition or a use of the name is searched for.
- The Display Language includes C macros and the pre-processor commands *#define, #else, #endif, #ifdef, #ifndef* and *#include*. The code must be expanded before the detailed name analysis can be performed by the thesaurus tool. This is provided by a script that calls *cpp* (C pre-processor).[1]
- The difference between the current version of the Thesaurus relation and the last generated data is detected by *diff*. This delta is used for update of the Versions_Thesaurus relation (see below).

[1] Analysing all files only once proved difficult since they are expanded by *cpp* each time they are included in another file. (They should only be analysed when it is their turn in the traversal of the directories.) However, *cpp* includes file name on the output so a special *awk* script was developed to exclude all the redundant cases.

- The *grep* command searches for patterns and is used to reduce the amount of code before it is handed to the *awk* scripts.
- The *sed* scripts are used for automatic string substitution – a means of massaging the code before it is analysed by the *awk* scripts.
- To prevent duplicates of names within the same container, the *sort* and *uniq* commands are used.
- The *sqlplus* command invokes Oracle with scripts containing SQL queries performing the following tasks:
  - i) generate the current version of the HMS schemata by querying the system catalogue;
  - ii) unload the contents of the thesaurus which is used for comparison (see below); and
  - iii) delete old contents of the thesaurus relations.
- The *sqlload* command provides the loading of the generated data into the database.
- A modified version of the query dictionary parser (a C program) analyses the query dictionary and generates the data to be inserted into the Thesaurus and Query_Dictionary relations.

The update of the Thesaurus relation is performed according to the following procedure. The contents of the current Thesaurus relation are unloaded to a file. The sequence numbers are deleted, and the file is then compared with the data just generated. A leading '<' on a line produced by *diff* indicates that the line is only in the generated data and should thus be inserted into the Thesaurus relation. A leading '>' on a line produced by *diff* indicates that the line is only in the Thesaurus relation and should thus be removed. The Query_Dictionary relation is updated in a similar manner.

The Versions_Thesaurus relation is also updated by using the output from *diff* mentioned above. The new data is inserted with an 'A' as the ADD_DELETE value. Instead of removing the disused names, as is the case with the Thesaurus relation, entries with a 'D' as the ADD_DELETE value are inserted.

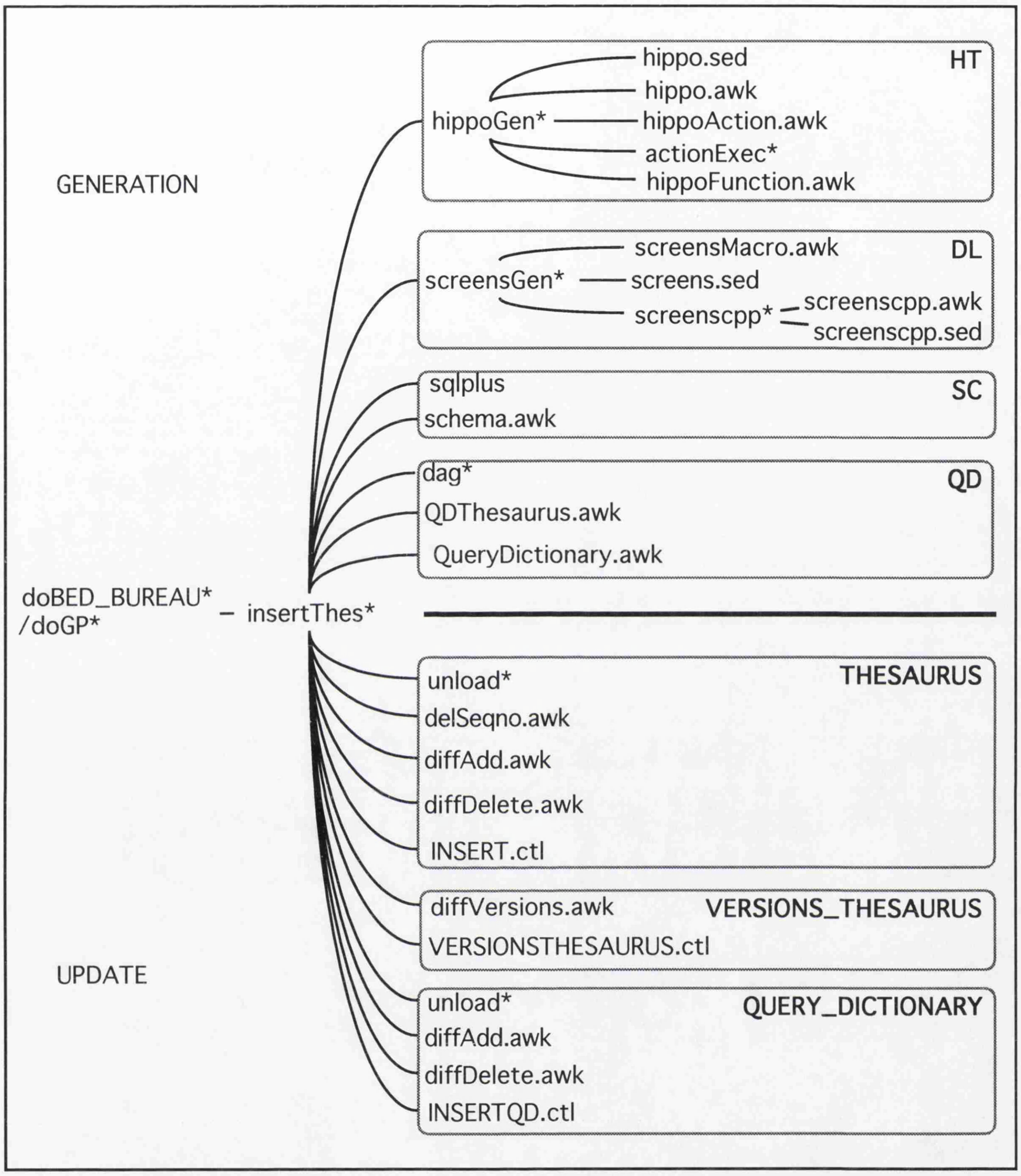


Figure 2.7: The thesaurus scripts and programs

Figure 2.7 describes the relationships between the scripts performing the data generation and the subsequent update of the meta-data relations. The extensions '*', 'awk', 'sed', and 'ctl' indicate C-shell, *awk*, *sed* and *sqlloader* control scripts, respectively. The upper part of the figure shows the scripts responsible for the data generation; the rounded boxes indicate a set of scripts for each of the container types HT, DL, SC and QD. The lower part shows the scripts responsible for the update.

2.2.4 Evaluation

The usefulness of the tool in the HMS project was evident after short time. For example, the database administrator used the tool to find all the QDfunctions using the various relations during a reorganisation of the database. The tool has also been successfully used for consistency checks and software reuse. This section also discusses other aspects influencing the success of the tool such as performance, how easy it is to learn and use the tool, and alternatives to the tool.

2.2.4.1 Detecting Inconsistencies

The thesaurus tool has proved useful in the process of finding bugs and inconsistencies in the HMS software. A few examples follow:

- Unused actions were found. The most common reason was that existing actions were replaced with new ones without the programmers remembering to delete the old ones.
- A few calls to non-defined actions were found.
- Inconsistent use of the macro commands *#ifdef*, *#ifndef* and *#include* was detected, e.g. C-shell environment variables not set as appropriate, non-existent files included (because the file names had been changed in the meantime), etc.

Some of the inconsistencies found by the tool (e.g. calling a non-defined action) might have been detected at run-time during a test. However, there will always be cases of bugs not detected in a test (they may occur when the system has been operational for half a year). In any case, it is advantageous to detect inconsistencies or bugs as early as possible.

2.2.4.2 Software Reuse

As in every large-scale application development project, one should aim at software reuse. The extent of reuse depends heavily on the information provided about the existing software and how the software is documented. As seems to be the case in most application development projects, in the HMS project there is hardly any written information available about software suitable for reuse, and the code itself is poorly documented. In the HMS case, the information is given accidentally and informally among the programmers. This may work as long as the project is small and there are only a few programmers, but in order to cope with growth in size and complexity, another strategy has to be chosen.

It is believed that the thesaurus tool will encourage code reuse. For example, the tool was helpful for the author in that several classes and screens, reused in the implementation of thesaurus interface, were detected via the thesaurus data.

However, to really benefit from the tool with respect to reuse, it should be extended with that purpose in mind. For example, one could introduce appropriate conventions for comments in the REMARK field of the Thesaurus relation. A problem is that programmers are generally reluctant to make comments. By introducing conventions, it would be easier for the programmers to make the comments, and they might become more informative.

2.2.4.3 Performance

The time it takes to generate and update the thesaurus data and the response times of the queries provided by the interface are two aspects of the thesaurus tool's performance. By July 1990 it took about 45 minutes to build the whole thesaurus for the BED BUREAU and GP applications. Performance is no problem since building is normally done overnight, and the cost of machine resources, at this stage, is effectively negligible. (The log in Appendix A shows how long it takes to build the various parts of the thesaurus.)

Some performance tests on the various queries of the interface have been carried out. There were approximately 4300 records in the Thesaurus relation and 1000 records in the Query_Dictionary relation at the time of the tests. Among the results were the following:

- The "Lookup" operation, retrieving between 4 and 70 records for various search strings, used between 5 and 13 seconds.
- The three "Change to X" operations, retrieving between 12 and 20 records, used between 3 and 15 seconds.
- The "Consistency Check" used between 3 and 31 seconds for the respective name types.

In general, response times depend of course on the actual computer, the machine load, the size of the thesaurus, the use of indices, etc. Nevertheless, since the thesaurus tool is not of the kind that people interact with continuously, the measurements presented above indicate acceptable response times.

2.2.4.4 Learning and Understanding

The effort needed to become an active user of the thesaurus tool can be divided into two. First, the user must learn how to invoke the features for *sort, look-up, change effects, consistency checks,* etc. This should be relatively straightforward since the interface is window-based with pull-down menus, buttons, help menus, etc. Second, and harder, is to understand the underlying model, which is necessary in order to interpret the results. That is, the user must:

- know the name types and container types;[1]

[1] This information is provided by the "*Information*" button.

- understand the distinction between definition and use of names; and
- understand how the predefined queries for consistency checks, change effects, etc. operate on the thesaurus data.

The knowledge and understanding mentioned above are directly related to the knowledge and understanding of the actual application system. That is, if the general understanding of the system is good, understanding and using the tool should be straightforward – and correspondingly difficult if the general understanding is poor. If the understanding is poor, it should be improved in any case. So, acquiring the knowledge needed for using the thesaurus tool should not be regarded as an unnecessary burden on the software builder.

2.2.4.5 Alternatives to the Tool

There are two principal strategies for solving the problems of finding the place of an action or a function call, the uses of a macro, the effects of schema change, etc. without the thesaurus tool. First, it is common that a few people are responsible for some part of the application, and they may feel they remember the software well enough to answer any question about the application. This might be possible as long as the application is small or is in an early phase of the development, but such a strategy is impractical in the long run.

Second, the system catalogue of the RDBMS and Unix commands like *grep* can be used. The system catalogue, however, yields only database specific information such as schema descriptions; it contains no information related to other parts of the application. *Grep* and other Unix commands can be used to search for names in application programs, but have several limitations compared with the thesaurus tool:

- It may be awkward to do *grep* on large applications with many (sub)directories. Experience shows that the risk of neglecting cases is significant.
- *Grep* cannot be applied to all parts of an application, e.g. not to the database schema.
- *Grep* may for example return the name of a query (say) 20 times in one Hippo program. A programmer can, of course, use the *sort* and *uniq* commands to remove duplicates, as does the thesaurus tool, but after all, the operations of the programmer cannot be as consistent and thorough as the operations of the tool.
- Since *grep* returns whole lines at a time, a lot of noise – irrelevant data – is included.
- Using *grep*, etc. programmers have to scan the whole application each time they need the information. This may take a long time and is inefficient with respect to programmer effort. In contrast, the thesaurus tool has the information already in its meta-database. Therefore, obtaining the information by executing predefined queries

on the meta-database takes less time and is more efficient. On the other hand, there is a small chance of the thesaurus' meta-data being out of date.

Regarding the consistency check and the impact analysis, their implemented queries are so complex that they are unlikely to be captured fully by *ad hoc* use of the *grep, uniq*, etc. commands.

2.2.4.6 Granularity of Container Types

It could be questioned whether the software builders would benefit from a finer partition of container types. For example, actions and classes could be specified as containers. There is a trade-off, however, between more detailed information and the risk of losing the overview of the thesaurus. For example, the notation for specifying the container of a name occurrence would be more complicated. Another consequence is that having a container type CL (class) should also imply having a container type OB (object). (Regarded as container types, there is no intrinsic difference between a class and an object). However, having an OB container type would be inconvenient in two ways. First, objects may be nested indefinitely (in theory). The reference to an object might therefore be an arbitrary long list of names. Second, due to the high number of objects (about 2000 in the BED BUREAU application by June 1990), the risk of losing the overview of the structure is significant.

2.2.4.7 Recording Change

The present version of the Versions_Thesaurus relation records only additions and deletions of name occurrences. Changes to the definitions themselves, i.e., the body code of the screen definitions (classes and objects in the Display Language), actions, functions, queries and update functions, etc. are not captured. Detecting and informing about such changes are complicated but would be very useful. One of the major problems of software development in teams is that one team member changes a definition, and the first time that another team member finds out about this is when his or her part fails.

A changed definition could be stored with a 'C' in the ADD_DELETE field[1] of the Versions_Thesaurus relation and the time for the incident in the INTRODUCED field as in the add/delete cases. In addition, the tool should ideally record the category of change. Proposing interesting change categories is not straightforward, but a few examples are: a new button referred to in an action, a field no longer referred to in an update function, an added join in a query, etc. An even harder problem is to detect the actual change. Depending on the requirement of the kind of change that should be provided, there are several options for presenting the differences between new and old versions:

[1] The field should be renamed accordingly.

i) present differences in the form of textual deltas;

ii) present differences in the form of new, removed or changed statements of the code; or

iii) present differences in terms of added or removed names within the containers.

The first option could be provided by, for example, applying the Unix commands *diff* or *comm* to the modified and old versions of the definitions of a screen, action, etc. Alternatively, the thesaurus tool could exploit delta information provided by version control systems such as RCS (see Section 3.3.1). Finding the semantics of the change would then be up to the user. With hundreds or thousands of screens, actions, etc. this would probably be an expensive task. By storing the code in the form of an abstract syntax tree or a similar structure, more detailed information about the kind of change in the code could be provided (second option). The first and second options would require the previous version of the code to be stored in order to produce the deltas and would thus imply major modifications to the present thesaurus tool.

If change information in terms of additions and deletions of name occurrences within a container is sufficient (third option), then the current Versions_Thesaurus relation could be used unchanged. That is, for a given container, information about added or deleted names of types corresponding to the container type could be provided. For example, changes to an action script could be measured in terms of added or deleted actions, datums, functions, queries, transactions or update functions. (Table 2.1 shows the possible name types for each container type.) If found convenient, the container types may be refined in order to provide more detailed information (see previous section).

2.2.4.8 The Tool in an Organisational Context

When introducing the thesaurus tool into an application development environment, one should pose the following questions:

- Who should use the tool?
- How should the working process be organised in order to benefit as much as possible from the tool?
- How should the project management motivate and encourage active use of the tool?

Concerning the consistency check feature, it is particularly important that inexperienced and immature programmers find bugs and inconsistencies by themselves before the software is released. The only purpose of the tool should be to improve the quality of the software; a negative attitude may be created if it is felt that the tool is used for individual monitoring purposes, such as by the project management.

2.3 Quantifying Evolution

By utilising the Versions_Thesaurus relation of the thesaurus tool, all changes (at present only additions and deletions) occurring within a given time interval (e.g. last year, month, week) can be found. The relation can thus provide information about the project development at different times. Usually, software engineers are too constrained by short-term goals to compile such statistics. One of the advantages of the thesaurus tool is that all data is generated automatically without the need of any user intervention. In general, the tool may be a helpful means for studying the behaviour of long-lived database systems [Atkinson 1990].

One area of system evolution that has been of particular interest recently is changes to database schemata (schema evolution). The effects of schema changes are divided into three categories:1

i) effects on other parts of the schema;

ii) effects on extensional data; and

iii) effects on application programs.

Typically, there will be many application programs that utilise a type that has been changed in the schema. These programs may use screen definitions, query definitions, procedures, etc. It is not difficult to imagine that incompatibilities between a schema type and the corresponding type assumed by the application programs may have serious consequences.

Recent work is concentrated in the area of object-oriented databases [Banerjee *et al.* 1987, Penney and Stein 1987, Skarra and Zdonik 1987, Kim and Chou 1988, Panel 1989, Lerner and Habermann 1990] where the consequences of changing a type (class) may lead to more significant changes in the schema itself than in a relational environment, but the consequences for extensional data and application code may be as serious as in a relational environment.

The purpose of this section is to present the results of an experiment that was conducted with the intention of quantifying the evolution of the HMS schema and quantifying the consequences of such changes on the rest of the application code. The period for the study started in June 1990 and continued until December 1991. Initially, the HMS system was analysed every fortnight, but due to repetitive changes to the development environment and because the author was not present to instantly adapt the tool to this kind of change, sustaining this frequency proved impossible (Section 2.3.3). All measurements until November 1990 were in the development period. Field trials began in November 1990. During the year from November 1990 to November 1991, the

1 Chapter 3 provides a more detailed discussion.

HMS system development continued with operational use in one hospital beginning in May 1991.[1] By December 1991 HMS was running in several hospitals. The project team grew from six to thirteen people during the period of investigation.

2.3.1 Evolution of the HMS Schema

During the period of study, the number of relations increased from 23 to 55 (139% increase) and the number of fields increased from 178 to 666 (274%). However, what is more interesting than this considerable growth in size, is that every relation has been changed. At the beginning of the development almost all changes were additions. After the system provided a prototype and later went into production use, there was no diminution in the number of changes, but the additions and deletions were more nearly in balance.

	Relations			**Fields**		
Date	**Added**	**Deleted**	**Current**	**Added**	**Deleted**	**Current**
22/6/90			23			178
6/7/90	6	0	29	103	0	281
20/7/90	13	0	42	78	0	359
3/8/90	1	-1	42	9	-15	353
17/8/90	18	0	60	97	0	450
Oct-90	3	-23	40	52	-126	376
Nov-90	47	-40	47	528	-376	528
Nov-91	40	-28	59	550	-290	788
Dec-91	20	-24	55	229	-351	666
Total	148	-116		1646	-1158	

Table 2.3: Added and deleted relations and fields in the HMS schema

Table 2.3 shows the development for the relations and fields. (A diagrammatic interpretation is given in Figures 2.8 and 2.9, respectively.) The number of deleted relations and fields appears as a negative value, so the *Current* value is the previous *Current* value plus the values of the *Added* and *Deleted* columns. *Added* and *Deleted* include both fields explicitly added to and deleted from a relation *and* fields added and deleted implicitly as a part of an addition or deletion of a relation. Most changes to the fields are such implicit changes. However, there are a substantial number of explicitly added and deleted fields as well. For example, of the 20 relations found in both the November 90 and November 91 schemata, only 4 have unchanged structure (the fields

[1] The operational system concerned the management of in-patient information. Many of the changes were the result of improvements to this system, changed requirements by government (the minimum data set) and the development of an out-patients system due for delivery in April 1992.

remained the same). During the period of examination, a total of 148 relations and 1646 fields have been added, whereas respectively 116 and 1158 have been deleted. That is, there have been 28% (relations) and 42% (fields) more additions than deletions.

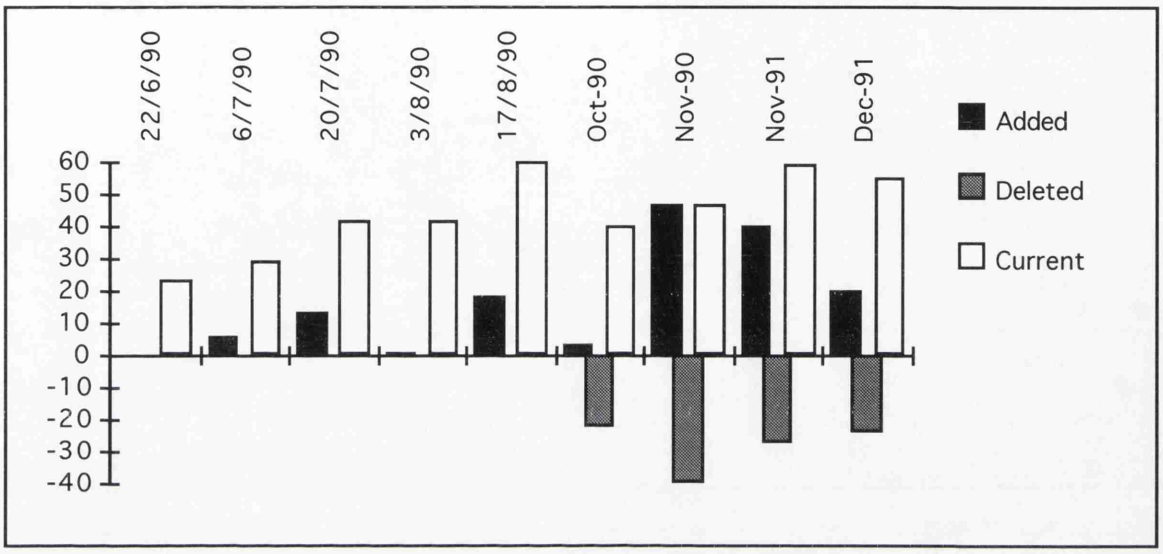


Figure 2.8: Change history of the relations

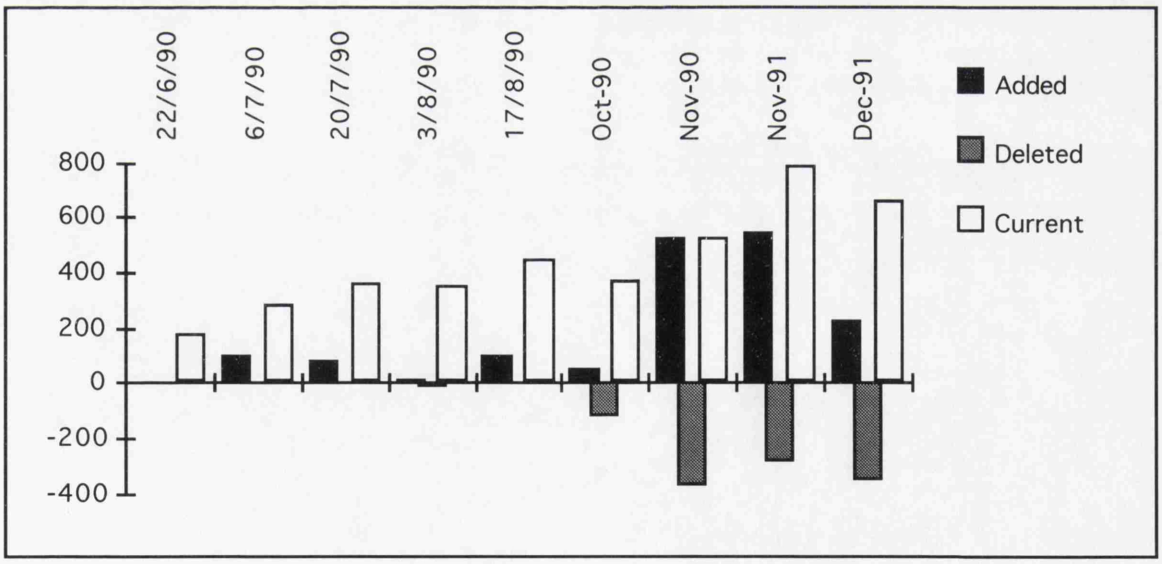


Figure 2.9: Change history of the fields

As mentioned, rename of a field or relation and changes to the type[1] of a field are not captured by the automatic measurements. However, a visual check on the November 91 and December 91 schemata found that there was only one rename of a relation where the

[1] A very general interpretation of the type concept is here used which includes the field properties unique, non-nulls, length and representation (*integer, char, date,* etc.).

relation's structure was unchanged, that 3 relations were vertically factored and that in one case 2 relations were joined together. The rest were "pure" additions and deletions. Regarding the fields, there were 18 renamings, 4 changes of unique/non-nulls, 23 changes of length and 4 changes of representation (3 from character to integer and one vice versa), i.e., 31 changes of field type. Respectively 31 and 48 fields were explicitly added and deleted.

In a large-scale project, with many people involved, there will always be different interests and opinions on how to solve the problems. Changes of the specification, context and customer generate drastic changes to the project. This was, for example, the case in the HMS project when the November 90 version replaced the October 90 version.

2.3.2 Consequences of the Schema Evolution

The previous section gives an impression of how significantly the HMS schema changed during the period of investigation. In order to provide a consistent application system, such schema changes have to be propagated to the application code. This necessary change propagation will be discussed in terms of the extent to which programs must be changed (edited) for each kind of schema change. The modification of the Nov-91 schema into the Dec-91 schema will be used as an example when describing the impact on the application code. A presentation of the use of the relations and fields in the Nov-91 version of the HMS system should help one to understand the example.

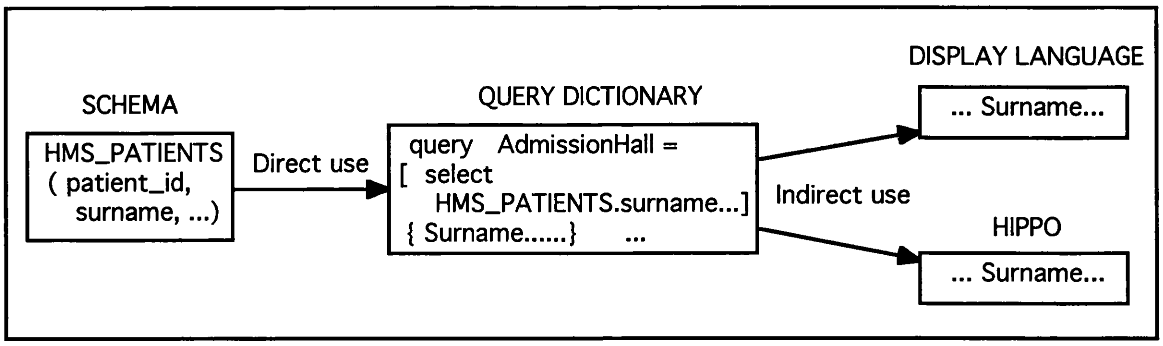

SCHEMA
HMS_PATIENTS (patient_id, surname, ...)
Direct use
QUERY DICTIONARY
query AdmissionHall = [select HMS_PATIENTS.surname...] { Surname......} ...
Indirect use
DISPLAY LANGUAGE
... Surname...
HIPPO
... Surname...

Figure 2.10: Direct and indirect use of relations and fields

Screens, actions, functions, queries, update functions, etc. are all dependent on the schema. The references to relations and fields in the screens and actions are all indirect via the query dictionary. The query dictionary was introduced to absorb change. Its analogy is a traditional view mechanism, but the query dictionary is more general supporting update and allowing interfacing to different DBMSs. Schema changes have direct consequences only for the query dictionary, but in general it is necessary to propagate these changes to the Display Language and Hippo code. For example, if the relation *HMS_PATIENTS* gets a new field, *place_of_birth*, the actual values must be entered via a screen (Display Language code). Furthermore, at least one application program should utilise this new information. Figure 2.10 illustrates the direct and indirect use of relations and fields. In the example, the query *AdmissionHall* uses the field *HMS_PATIENTS.surname* whose value is assigned the datum *Surname* which in turn is used in Display Language and Hippo code.

Measurement	**Number**	**Min**	**Max**	**Mean**	**Std**	**Sum**
Relations	59	0	101	16.9	27.1	997
Fields	788	0	167	6.6	14.2	5181
Fields grouped by Relation	59	0	795	87.8	178.3	5181

Table 2.4: Direct use of relations and fields in the query dictionary

Table 2.4 describes the direct use of relations and fields in the query dictionary. The first measurement, “Relations”, shows that among the 59 relations there is at least one that is never used (*Min*) and at least one other used 101 times (*Max*). The average is 16.9 (*Mean*), and the total number of times a relation name appears in the query dictionary is 997 (*Sum*). The standard deviation (*Std*) is high because most of the use is represented by only a few relations.

Both the “Fields” and “Fields grouped by Relation” measurements describe use of the fields. The extra information obtained by introducing “Fields grouped by Relation” is that the field statistics are related to the associated relation. For example, the maximum value 795 in row of “Fields grouped by Relation” indicates that there is at least one of the 59 relations that has in total 795 occurrences of its fields. An analysis of the raw data reveals that the fields of 3 relations constitute 45% of the use which implies a high standard deviation. The maximum number of occurrences for a field is 167; the average 6.6. The total number of field occurrences in the query dictionary is 5181.

Measurement	**Number**	**Min**	**Max**	**Mean**	**Std**	**Sum**
Fields	788	0	193	5.0	14.0	3946
Fields grouped by Relation	59	0	661	66.9	152.5	3946

Table 2.5: Indirect use of fields in Display Language and Hippo code

Table 2.5 shows the indirect use of fields in the Display Language and Hippo code. These measures have been obtained by:

i) finding all correspondences between fields and datums in the queries and update functions,

ii) finding all the queries/update functions[1] and datums used in the Display Language or Hippo code, and

iii) joining the results of i) and ii) by query/update function and datum.

The 788 fields are on average used indirectly 5.0 times, whereas the measure for fields grouped by relation is 66.9 times. The use of a field and all fields of a relation ranges from 0 to 193 and 0 to 661 occurrences, respectively.

As an illustration of consequences of schema changes, the effect of the modification of the Nov-91 schema into the Dec-91 schema is now described. Figure 2.11 shows that more than one third (36%) of all name occurrences had to be deleted. There were only a few renamings (less than 1%). The consequences of adding relations and fields are difficult to measure, but the minimum number of necessary additions can be estimated to about 10% of the number of existing name occurrences (see discussion below).

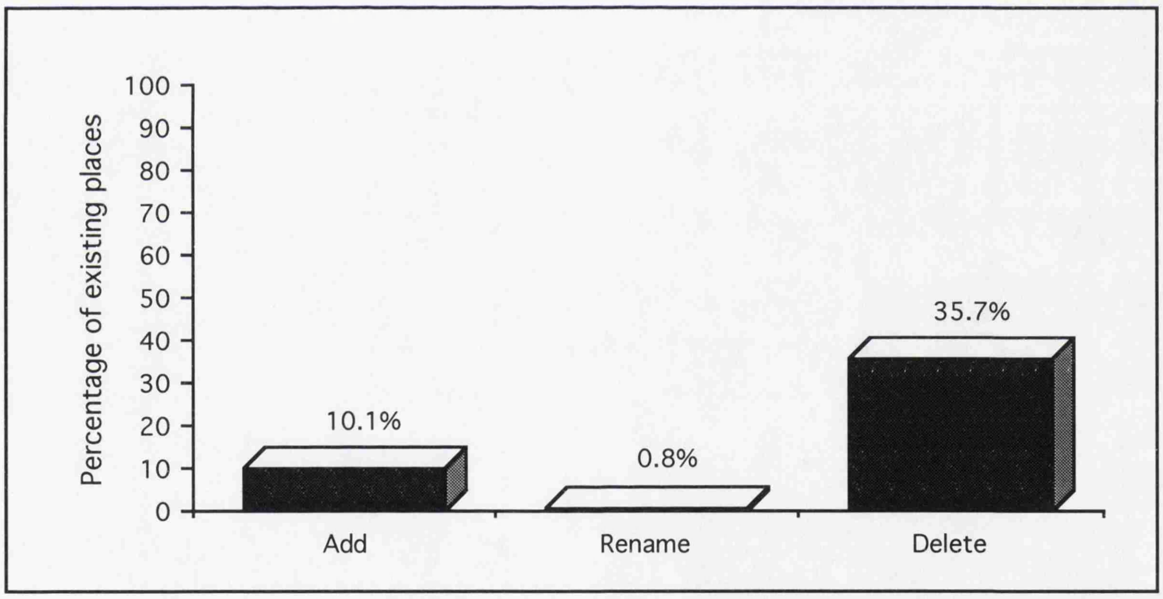


Figure 2.11: Consequences of the December 1991 HMS schema modification

A more detailed description of the consequences is given in Table 2.6 which contains one row for each kind of schema modification. (The number in brackets is the number of occurrences of the named change.) The change consequences are measured in terms of how many *places* that need to be edited for the changed relations and fields. A place is a position in a query or update function where a relation, field or datum name occurs, or

[1] A transaction call is regarded here as a call to all its containing update functions.

where a datum name in a Display Language or Hippo program occurs.[1] Duplicates have been removed. That is, the measurements record only one occurrence of a relation, field or datum name in each container. (In the actual code there are about twice as many occurrences.) In Table 2.6 *Query Dictionary* means queries or update functions and *DL or H* means Display Language or Hippo programs.

	Query Dictionary			DL or H	
Operation (occurrences)	**Relations**	**Fields**	**Datums**	**Datums**	**Total**
Add relation (19)	38	360	360	360	1118
Add field (31)		62	62	62	186
Rename relation (1)	8				8
Rename field (18)		128			128
Delete relation (23)	268	1555	628	1370	3821
Delete field (48)		351	151	156	658
Total (140)	314	2467	1201	1948	5930

Table 2.6: Consequences of the December 1991 HMS schema modification

For each added field at least one screen (Display Language code) should collect the new data, and an update function should insert it into the database. Moreover, at least one Display Language or Hippo program should eventually use the new data which also implies a new or modified query. To collect and use the fields of an added relation, the argument above implies that the relation name must be included in an update function and query as well. So, the names of the 19 added relations in the Dec-91 schema[2] must be inserted into the query dictionary at least 38 times. These relations have 180 fields implying that a minimum of 360 places for the fields and 360 places for the corresponding datums must be edited in the query dictionary and at least the same number of datum names in the Display Language or Hippo code. It is generally impossible for a tool to detect places affected by additions. Human intervention is required.

The renaming of the single relation and the 18 fields cause at least 8 and 128 places to require editing. There is not necessarily any effect on the Display Language or Hippo code because the name change may be absorbed in the query dictionary. However, if the intention is that new field names should be propagated to the corresponding datums, then 97 datums in the query dictionary and their 112 uses in the Display Language and Hippo code would also have to be edited (not shown in Table 2.6).

[1] A place could be localised by, for example, a (line number, word number) pair.

[2] Table 2.3 shows 20 added relations (not 19) because the single renamed relation is registered as one deletion and one addition by the thesaurus tool.

An examination of Table 2.4 reveals that removing a relation will on average affect 87.8 field occurrences in the query dictionary. In the best case, no field occurrences will need to be edited, but 795 in the worst case. The average number of field occurrences of the 23 actually deleted relations is 67.6, indicating that these relations are used less than average. The consequences of the deletions, however, are still significant. The deleted relations cause 268 removals of the relation names and 1555 removals of the names of their fields. These field names correspond to 628 datums, which have 1370 occurrences in the Display Language or Hippo code. In summary, the number of places affected by the deletion of the 23 relations is 3821.

In addition to the changes described above, some new update functions and queries will generally be needed which may be referenced in the Display Language or Hippo code. However, the query dictionary may absorb such changes because the same update functions and queries can operate on new relations and fields with only internal changes. That is, their references in Display Language and Hippo code may be unchanged. So, introducing a query dictionary is one means of alleviating the consequences of schema changes.

This section has described consequences of schema modifications in terms of the number of changes to places in which names of relations, fields and datums occur. It should be emphasised that the actual number of changes to the code will be much larger. For example, if a relation is added, not only will the names of the relation, its fields and corresponding datums be added (at least twice), but other code segments related to these items will be added as well.

2.3.3 Problems of Measuring Evolution

The thesaurus tool was installed to measure the changes to the HMS schema and its consequences over the 18 month period from June 1990 to December 1991. However, in addition to the changes to the HMS schema and application programs, the system structure and development environments also changed significantly (mainly to cope with the growth of the system). The thesaurus tool itself had to be changed correspondingly.[1] The kinds of change were:

- Completely new structure and names of directories and change to file name conventions.
- Changes to the support software (operating system, DBMS, version control systems, etc.).

[1] One year after the initial release of the thesaurus tool, the author was asked by the HMS development team for a total upgrade of the tool so that it could cope with, and benefit from, the changed environments.

- Changes to the application programming languages, like modified syntax and extended run-time library (the query dictionary language, Display Language and Hippo language were all changed during the period of investigation).

Keeping the continuity of the observations may prove difficult due to such changes.[1] As mentioned, they were the reason for the different time intervals shown in Table 2.3.

Anybody attempting to carry out similar experiments or build equivalent tools would certainly need to cope with changes in the representation and storage organisation of the software and new versions of programming support software. In the HMS system the program directories were reorganised without notifying the thesaurus tool. (Figure 2.12 illustrates the scale of change.) This excluded several programs from the analysis for a short period of time. Another failure was that the program for unloading the thesaurus data from the database was not recompiled when a new version of Oracle was introduced. The result was that no data was unloaded. The tool then assumed (wrongly) that the thesaurus relations were empty, and the subsequent test for change detection was invalidated. Therefore, thesaurus tools need to be subject to the same change control mechanisms as the rest of the system under study.

Major changes to the languages used may be unusual in a typical programming environment. Nevertheless, the HMS application languages are continuously being developed, and this was experienced as a problem during the implementation of the thesaurus tool. New constructs, new built-in functions, etc. in the application languages had to be reflected in the thesaurus analyser. For example, at present, *awk* scripts test potential actions and functions against a list of built-in names. All changes have to be done manually with the risk of being insufficient or not performed at all (at least in a transitional period). A more sophisticated version of the tool should be driven from the same data as the parsers of the respective languages, enabling changes to be automatically reflected in the thesaurus analyser.

Completely automated collection of change data seems impossible. Therefore, in order to collect reliable measurements of a real-world system, the application development people on the site must have the time and interest in co-operating with the experiment. One problem is to convince them that the data collection is worth the investment. This problem may not be so great if the change measurement and management tools were closely integrated with the programming environment.

[1] In the HMS case, the hardware was at least the same throughout the period of investigation. Hardware changes may result in yet another category of problems. For example, since the period of investigation a major part of the work has moved from Unix machines to PCs under Windows 3.

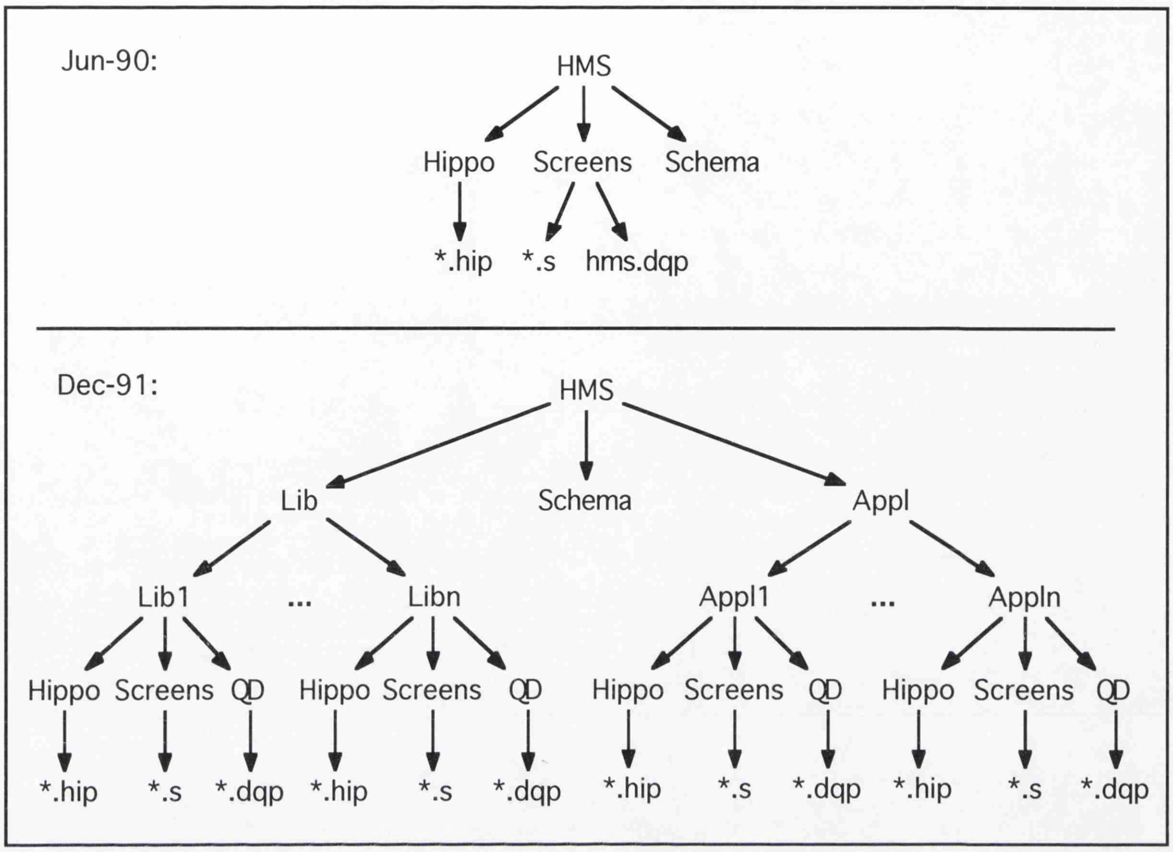


Figure 2.12: Extension of the system structure

2.3.4 Schema Evolution in Different Application Domains

The purpose of collecting change measurements is to identify requirements for methodologies and tools for maintaining large, long-lived application systems. The measurements presented in the previous sections were conducted to discover the extent of change to the schema of a health management system. The results show that, at least in this application, there were large numbers of changes with considerable consequential changes to code. However, to acquire more general knowledge about the extent and form of change, applications systems in various application domains should be measured.

In recent literature on schema evolution, which mostly concerns object-oriented databases, it is commonly claimed that facilities for schema evolution are more important in new application areas such as CAD/CAM, CASE design, etc., than in traditional areas such as payroll, accounting, reservation systems, etc. These claims are generally not supported by measurements. On the other hand, a study of the evolution of seven traditional applications ("Sales and payments", "Property inventory", etc.) [Marche 1993][1] shows that approximately 60% of the entity attributes changed during the period

[1] Marche's study and a summary of the measurements presented in this chapter [Sjøberg 1993], are the only examples of schema (or data model) evolution measurements found in the literature.

of investigation. The data models[1] were measured (manually) in a period varying from 6 to 80 months for the respective applications.

In spite of the changes reported in that study, it might be the case that there are fewer schema changes in traditional application domains than in newer areas. One reason could of course be that there is less need for change in traditional systems because they are simpler and their functionality and behaviour better understood. Another reason, however, could be that the traditional systems are so rigid, and the consequences of change so enormous, that due to lack of appropriate methods and tools, user requested change is simply rejected. An example is the Norwegian census database – a CODASYL network database containing 5 gigabytes of data about 5 million persons. In spite of changed user needs, the schema has not been changed during the last decade due to extreme costs – typically measured in units of person-years; any (minor) schema change would imply database reorganisation and application code modification, the needed work amounting to at least half a person-year per minor schema change [Gløersen 1993].

2.4 Summary

Research on methodologies and tools for change management should focus on real-world problems. In accordance with this view, the problems of a health management system (HMS), currently running in several hospitals in the UK, were investigated in detail. HMS is implemented as a relational database with application programs written in different languages tailored for defining and using screens, actions, functions, queries, update functions, etc. When the system reached a certain size, it was evident that aids in keeping track of the structure of the system were becoming essential. The thesaurus tool was therefore developed. The tool is centred around the thesaurus which is a meta-database containing information about names and identifiers defined and used in all the software including database schemata. For each name occurrence, the thesaurus records the name type, the container of the occurrence, the container type, whether the occurrence is a definition or a use, and so forth.

It should be emphasised that the thesaurus spans all the languages used in the HMS system, which is in contrast to, for example, the system catalogue which only contains database specific information such as schema descriptions. Hence, the thesaurus tool also tracks down dependencies across software written in different languages, for example dependencies between fields used in queries and variables used in screens and actions. All the thesaurus information is automatically generated, and the thesaurus is regularly updated (every night).

1 In compliance with Marche's parlance, *data model* means in this context a concrete model of an application and should thus not be confused with data models such as the relational data model [Tsichritzis and Lochovsky 1982].

The thesaurus tool has been successfully used in the HMS project. The usefulness is closely related to scale. So, as the HMS system grows and becomes more complex, with many new programmers entering the project, the tool is expected to become even more useful.

The thesaurus tool also contains change history in terms of added and deleted entries. This information was used in a study conducted with the purpose of quantifying schema evolution and its impact on the rest of the application. The HMS system was observed over a period of 18 months. Both the schema changes and their consequences, measured in terms of potentially affected places in the application code, were significant.

Most of the recent research on schema evolution has focused on object-oriented databases. Ideas for managing the impact of schema changes on the schema itself (class hierarchy) and on extensional data (objects) have been implemented. Managing the consequences for application programs (methods) proves to be a more complex issue. The results reported in this chapter were based on the use of a relational DBMS and confirm that change to database schemata is an important issue – independent of the data model of the actual application – and that change management tools are needed – at least in the context of advanced and experimental application development such as that measured here. The extent and sort of change may differ between various application domains. Others are invited to conduct similar experiments, based on similar technology, in other environments. Much effort is required, however, to carry out such experiments and may be one reason for the lack of schema evolution statistics reported in the literature.

This chapter has discussed the thesaurus tool which was developed due to lack of appropriate commercial tools. The next chapter surveys existing and proposed software engineering techniques and tools for use in commerce and industry.

Chapter 3

Software Evolution and Supporting Tools – A Survey

3.1 Introduction

This chapter describes concepts in the field of software engineering with emphasis on supporting change. State-of-the-art models and tools for change management will be discussed as a context for the research presented in the subsequent chapters.

3.1.1 The Software Development and Maintenance Process

Software engineering includes models, methodologies, techniques and tools for system construction and maintenance. The classical model for describing the software development process is the so-called *waterfall model* [Royce 1970]. This analysis-design-implementation-testing model of the software development life cycle, however, does not comply with the way systems are built in the real world. Obvious inadequacies are the lack of recognition of the importance of system changes and its description of system development as a sequential process. The *spiral model* [Boehm 1988] was introduced to encompass some of the deficiencies of the "waterfall model". The "spiral model" adds the notion of risk analysis and allows for iteration of the development tasks.

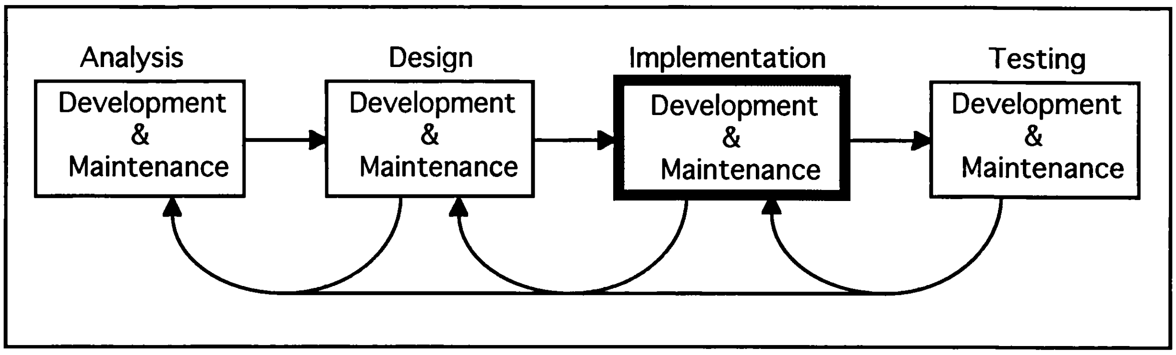

Analysis
Development & Maintenance
Design
Development & Maintenance
Implementation
Development & Maintenance
Testing
Development & Maintenance

Figure 3.1: The software development and maintenance process

The problem of software maintenance, however, is not explicitly addressed by any of these models, though it is common to extend the classical model with a separate maintenance phase after testing [Sommerville 1992, Pressman 1992]. Instead of representing maintenance as a final phase after testing, each box (phase) in the simplified model of Figure 3.1 contains elements of both development and maintenance. In practice, the phases of development are repeated during maintenance. New requirements must be determined, the existing software application needs re-design and re-coding, new tests must be undertaken, etc. This complies with the view presented in [Lehman 1980] that both development and maintenance should be regarded as one process – *software evolution*. This does not mean that the software development and software maintenance life cycles follow the exactly same pattern; at a detailed level the stages and the relative effort applied to the stages may differ [Chapin 1988]. Nevertheless, a detailed discussion of the suitability of software process models is not a concern of this thesis. The intention of this thesis is to describe tools and techniques for managing various kinds of software change – independent of whether they occur during initial construction or after the software application has become operational.

The purpose of this introduction is to place, in terms of life cycle phases and levels, the concepts (processes, tools and tool environments) to be discussed in this chapter. The emphasised box in Figure 3.1 indicates that implementation aspects will be the focus. Figure 3.2 presents a more detailed picture. The boxes with thick borders indicate concepts directly related to the research presented in this thesis. The thin border boxes represent concepts that are included merely to illustrate the context of the concepts under consideration. A concept placed in the implementation phase may also be relevant to other phases (for example, a data dictionary may contain design information). From the viewpoint of this thesis, however, the concepts are interesting for aspects relevant to implementation. The figure shows two levels. The lower level describes concepts pertinent to particular phases, whereas the upper level describes concepts concerned with the management of the overall software construction and maintenance process.

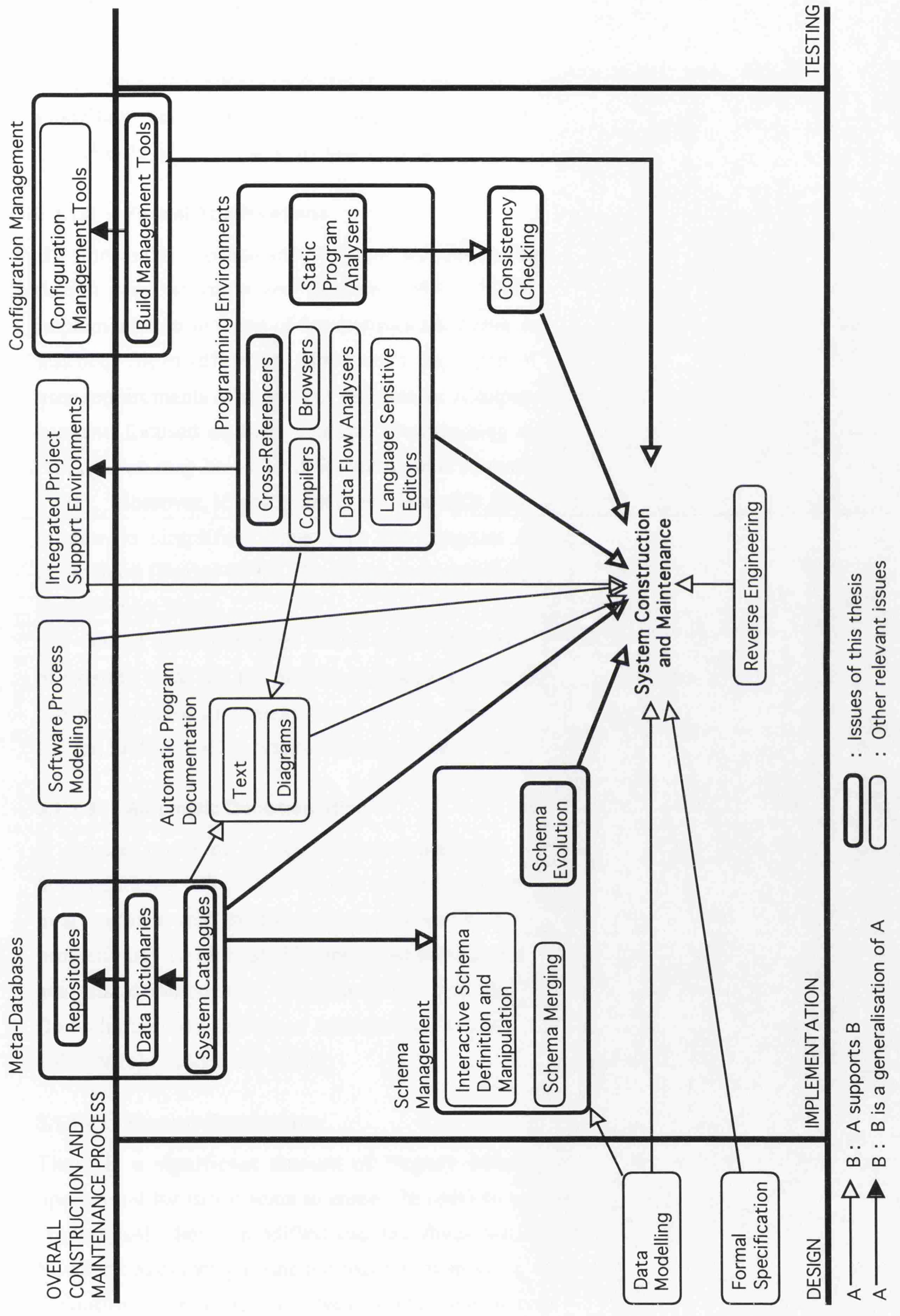


Figure 3.2: Concepts in software construction and maintenance

3.1.1.1 Data Modelling

A data model is characterised by having an inherent structure and a set of techniques and tools used in the process of designing, constructing and manipulating databases. The data modelling process produces, among other things, database schema specifications and is thus closely related to schema management.

3.1.1.2 Formal Specifications

By introducing formal methods and corresponding tools the quality of analysis and design may be improved [Bjørner 1991]. This may in turn amend the software implementation in terms of fewer errors and better structure. Consequently, there will be less need for modification. However, a major part of software change is due to change in user requirements after system installation (Chapter 1). Work on formal specifications has also focused on that problem. Determining change effects and necessary change propagation may be easier if the software is formally specified [Nakagawa and Futatsugi 1991]. Moreover, if a program is automatically generated from a design, the maintenance process is simplified since only the program specifications need to be manually maintained [Baxter 1992]. However, a practical realisation of this approach is probably several years away.

One of the deficiencies of this approach is its ability to cope with the large quantity of existing code (cf. the problem of "legacy systems" [Brodie 1992]). How should one formally specify all those systems whose implementations generally are a mixture of code written in different languages and data stored in various forms?

3.1.1.3 Automatic Documentation

A severe problem in the software application industry is obsolete or missing documentation. The major reason for this is that documentation is normally not updated in accordance with modifications to the software. For some sorts of documentation this problem may be alleviated by tools that provide automatic documentation based on static analysis of source code. Such tools may typically generate call graphs, control and data flow charts, cross-reference information, metrics reports, etc. [Ryder 1979, Jandrasics 1981, Meekel and Viala 1988].

3.1.1.4 Reverse Engineering

There is a significant amount of "legacy code" [Brodie 1992] which will still be operational for many years to come. In order to satisfy new requirements, such code is continuously being modified causing deterioration of its structure [Lehman 1978]. Reverse engineering is one approach to help solve this problem [Bachman 1988]. "By definition, re-engineering changes the underlying technology of a system without affecting the functionality" [Colbrook and Smythe 1989]. The idea of reverse engineering is to generate an abstract version of a concrete program and then re-implement the abstract version. Since most existing code is written in COBOL, typically COBOL programs are being re-implemented in COBOL itself or in a more up-to-date programming language. Even though almost all reverse engineering tools are developed for COBOL, recent work on restructuring is also reported for other languages [Griswold and Notkin 1992].

3.1.2 Change Management – An Aspect of Project Management

Project management is an activity at the overall software life cycle level and involves tasks like scheduling, team management and resource allocation (people, programming languages, tools, operating systems, hardware, etc.). The administration of changes at this level is an important part of project management and is commonly referred to as *change management* [Humphrey 1989, IBM 1992].[1] The change process is formalised in that all change requests are evaluated with respect to the need for the change, the impact of the change on the project and system, schedule of necessary activities, etc. During the implementation of a change, information is recorded about who did what when, what is the status, what remains, etc. IBM's Information/Management product is an example of a tool providing support for such change management [IBM 1992]:

> The Change Management facility helps you coordinate the various tasks your organization performs to make and test changes in your data processing environment. You can enter data about changes made to any area of your organization's operations: to software and hardware components of the operating system or to procedures, publications, and facilities.

Change management tools at this level are thus support systems that record information and produce corresponding reports.

3.1.2.1 Software Process Modelling

One approach to managing changes is to describe software processes by programs written in a *software process programming language* [Osterweil 1987]. A collection of such programs may constitute a formal model for a particular process typically involving activities like editing, compilation, change of tools, etc. and objects such as specifications, tools, hardware, etc. It has been argued that given a formal process model, impact analysis and propagation of necessary consequential changes may be automated [Sutton *et al.* 1990, Shepard *et al.* 1992]. An example is: "What are the consequences of changing a programming tool?"

The feasibility of describing software processes formally has still to be investigated. At least, there will be many aspects that can never be captured by such a formalism.

1 Another term, which seems to be used synonymously, is *change control* [Ferraby 1991, Pressman 1992]. Yet another related term is *configuration management* which will be discussed in Section 3.3.

3.1.3 Software Change Management – Focus of this Thesis

The focus of this thesis is methodologies and tools for change management at the software *implementation* level. They should support the software builder or maintainer in the following activities:

- Predicting change consequences (impact analysis)
- Propagating necessary consequential changes
- Detecting inconsistencies after change or preventing them
- Detecting and recording change (necessary for recompilation, etc.)

An issue is to what extent it is possible to automate these activities.

A traditional software application can be viewed at two levels of granularity. At the coarse granularity, the application is viewed as a collection of files or programs, and databases. At the fine granularity, the application is viewed as a collection of concepts like type definitions, variables, values, procedures, statements, etc. depending on the actual programming language. That is, the *contents* of the files or programs are important. This distinction corresponds to the two tiers in the Eclipse Two-Tier Database [Cartmell and Alderson 1989]. First tier data is *objects* (files, directories, records, etc.); second tier data is *fine structure objects* (sentence in a document, nodes in an abstract syntax tree, etc.).

At the fine granularity level, a particular kind of change that may have serious consequences for the rest of the application, is change to type definitions or schema evolution in a database context. Determining the effect of type or schema changes and ensuring correct change propagation is a challenging research issue.

Change management at the coarse granularity level involves problems such as keeping track of which versions of a program can be integrated with which versions of other programs in order to constitute a valid application (configuration management). A special case of configuration management is when only the last (i.e., current) configuration is managed. This is commonly referred to as *build management*. Re-building typically involves recompilation and relinking; the classical supporting tool is Make [Feldman 1979]. In order to determine dependencies between programs, the program contents (fine granularity level) must be investigated. For a programming language like C, with simple dependency relations between programs, there are several tools that automatically determine such dependencies. A research issue is how to develop similar tools in other, more sophisticated programming environments.

A consequence of software changes is that the application system may become inconsistent. In this thesis *consistency* means compliance with the rules or constraints of a given *application independent* model.[1] A constraint could, for example, be that nothing should be declared without being used. The consistency term should thus not be confused with *application dependent* constraints such as those specified in an application model developed with a data modelling tool [Cooper and Qin 1992]. However, the same technology may be used to express constraints of both kinds, cf. work in the context of ADABTPL [Fegaras *et al.* 1989, Fegaras and Stemple 1991] and TRPL [Sheard 1990].

3.2 Schema and Type Evolution

One of the most challenging problems of constructing and maintaining large, long-lived data-intensive application systems is to cope with all the changes that inevitably will be imposed on the systems over time. Many large application systems are centred around a database, and a particular kind of change that may have serious consequences for the rest of the application systems (Chapter 2) is change to database schemata – schema evolution.[2] User requested enhancements in functionality are a major cause of schema evolution. Modifications to schemata or type definitions may also be consequent on merge of database applications. The respective schemata need to be integrated [Batini *et al.* 1986] requiring resolution of naming conflicts, removal of duplicates, determining dependencies between definitions, etc.

At present, schema modifications are often dealt with in an *ad hoc* way. The necessary data conversions and program modifications may be expensive due to factors such as a requirement to shutdown the system, programmer effort, machine resources, etc.

Research on schema evolution is still in its infancy. Some recent work has been undertaken in the context of relational systems [McKenzie and Snodgrass 1990, Ariav 1991, Roddick 1992], but the majority has been concentrated in the area of object-oriented databases [Banerjee *et al.* 1987, Penney and Stein 1987, Skarra and Zdonik 1987, Kim and Chou 1988, Lieberherr and Holland 1989, Osborn 1989, Panel 1989, Lerner and Habermann 1990, Casais 1991, Waller 1991, Zicari 1992, Bratsberg 1993, Monk and Sommerville 1993].

Some parts of the literature give the impression that the problem of meta-data changes is particular to application areas such as computer-aided design and manufacturing (CAD/CAM), computer-aided software engineering (CASE), etc. Moreover, since the database research in this field has concentrated on object-oriented

1 However, the model will typically depend on the actual programming language or other aspects of the programming environment.

2 Schema evolution in traditional databases corresponds to class evolution in object-oriented database systems, to type evolution in applications developed in strongly typed programming languages and, at a higher level, to changes to conceptual data models.

databases, the problems of schema evolution have been associated with object-oriented technology. For example, in [Banerjee *et al.* 1987] it is claimed that:

> ... existing database systems allow only a few types of schema changes: for example, SQL/DS allows only dynamic creation and deletion of relations (classes) and addition of new columns (instance variables) in a relation ... the applications they support (conventional record-oriented business applications) do not require more than a few types of schema changes ...

This is not necessarily true, as was demonstrated by the study described in Chapter 2 and in [Sjøberg 1993] showing that the kinds and extent of schema changes may be significant also in areas where relational systems are used.

Nevertheless, it might be the case that the schemata are modified more frequently in special design support applications than in conventional "record-oriented business applications" (see discussion in Section 2.3.4). And, since the schema structure of object-oriented schemata is typically more complex than relational schemata, the consequences of changing a type (class) in an object-oriented environment may lead to more significant changes in the schema itself than in a relational environment. However, the consequences for extensional data and application code, which depend on the volume of data in the database, the amount of application code based on the schema, etc., may be as serious as in a relational environment. The effects of schema changes are divided into three categories:

i) Effects on other parts of the schema

ii) Effects on extensional data (user data stored in the database)

iii) Effects on application programs (including interfaces for data entry, queries, report generation, etc.)

The respective categories are discussed in the following sections.

3.2.1 Consequences on other Parts of Schema

In a strongly typed world, a change to one type definition may affect other type definitions in which the former is used. Naming conflicts may occur when new type definitions are created.

A relational schema offers little support for ensuring consistency; only relationships between a relation and its fields are captured. For example, an RDBMS does not normally provide mechanisms for expressing that a field is a foreign key (cf. referential integrity). A relational schema has a simple structure with poor modelling capabilities, and the kinds of schema change are correspondingly few (see Section 2.2.2.2 for a list of logical schema changes). So, hardly any schema change is consequent on other schema changes (except in cases of naming conflicts). This does not imply, however, that it is simple to perform schema changes in a relational environment. On the contrary, it means that consistency of the extensional data and the application programs must be ensured by the user almost without any support from the system itself.

In an object-oriented environment, a change in one place in the class hierarchy may have significant impact on other parts of the hierarchy. A schema change taxonomy is presented in [Banerjee *et al.* 1987]. That paper also describes invariants for ensuring consistency after schema changes and rules for guidance in cases where there is more than one way to preserve the invariants (rules for solving name conflicts, the rule that the domain of an instance variable can only be generalised, not specialised, etc.).

3.2.2 Consequences on Extensional Data

The need for evolution is still present after the database has been populated with user data. Addressing the consequences of schema change on the extensional data is therefore required. If a type definition is changed, the instances of that type must conform to the new definition.

3.2.2.1 Conversion

A typical consequential change of schema evolution is database reorganisation [Sockut and Goldberg 1979]. Ideas for avoiding reorganisation after schema change by introducing views have been proposed [Tresch and Scholl 1993], but only special cases of schema change can be accommodated. For example, a view mechanism does not support augmented information capacity such as adding a new attribute, and it is often not possible to create new instances of a view.

A conversion strategy can be *immediate* in that all instances are converted in one "big bang" when the schema is changed, or it can be *lazy* in that the instances are changed only when they are needed (accessed).

3.2.2.2 Filtering

An alternative to the conversion strategy described above, is a strategy where different versions of a type definition coexist. Under such a scheme "every instance of the type remains linked to the version under which it was created" [Skarra and Zdonik 1987]. The user can specify filters to make *type changes transparent* [Ahlsen *et al.* 1983]; that is, old instances can be viewed as instances of the new type and thus be used in application programs conforming to the new type. Similarly, new instances can be viewed as old instances and thus be used in existing application programs.

The usefulness of this strategy still has to be evaluated; there are practical limitations, particularly in large systems.

3.2.3 Consequences on Application Programs

The literature reports little research on the impact of schema evolution on the existing application programs. This is in stark contrast to its significance for application programmers (Section 2.3.2).

Typically, there will be many application programs that utilise a type that has been changed in the schema. These programs may use screen definitions, query definitions, procedures, etc. It should not be difficult to imagine that incompatibilities between a schema type and the corresponding type assumed by the application programs may have serious consequences. For example, if a new information carrying capacity is added to the schema, programs that do not use it should not change. However, at least one program must be created or changed to collect the data, and all programs that display closely related data should be considered for amendment to show the new data. This will in turn propagate to new screen designs and changed working practices.

3.2.4 Approaches

A strategy based on subtyping may support limited forms of evolution [Wegner and Zdonik 1988].[1] It allows for adding attributes to a type (which is a common kind of change). No change to other types is necessary, and there is no need for conversion since there are no instances of the new (sub)type. In an object-oriented database system this strategy may allow code to continue in operation but may reduce the information to programmers about where consequential changes are necessary in the programs that should utilise the change.

A combination of strategies based on views (allowing changes for preserving or reducing the information capacity) and subtyping (allowing changes for augmenting the information capacity) may cover many kinds of change. However, to the author's knowledge no evaluations of such strategies in the context of real-world applications have been reported. With a plethora of versions of each type (many implementations of the same conceptual type) there is a risk of creating unmaintainable and inefficient code.

Most of the research on schema evolution has been undertaken in an untyped world. A major and challenging research issue is to what extent schema evolution and its consequences can benefit from a strongly typed world that allows for persistent programs and other kinds of data. This problem is not investigated in depth in this thesis, but advisory systems have been developed, and the methodologies and tools to be described establish a framework for further research (Section 8.2.1).

1 An unsolved problem, first identified in [Albano 1983], concerns subtyping in combination with assignments. A general discussion of this problem can be found in [Connor *et al.* 1991].

3.3 Software Configuration and Build Management

Software configuration management is a discipline for controlling change and managing software modules that have been subject to change. Configuration management tools assist in controlling *versions* of the modules and in building *configurations* of a system. A configuration is a collection of all the modules of a system where each module is represented by exactly one version selected according to a certain criterion (e.g. the latest version).

3.3.1 Source Code Control – SCCS/RCS

SCCS [Rochkind 1975] and RCS [Tichy 1985] are two classical configuration management tools which manage multiple versions of text files (typically program sources). Files are read, compiled and edited according to a check-out/check-in protocol. For storage efficiency, a change to a file is recorded as a delta. To create the latest version of a system, SCCS applies all the deltas to the original version. RCS applies such forward deltas only to branches (which represent variants of a version) but apply reverse deltas to versions, which gives fast access to the latest versions. Identifying valid configurations and keeping track of them is up to the user, but support is given in that the tools automatically generate and manage the version numbers. RCS also allows for symbolic names, enabling combinations of versions to be described independently of version numbers.

3.3.2 Build Management

3.3.2.1 Make

One kind of configuration is software modules combined into complete executable programs. Such configurations are built by compiling and linking modules. Make [Feldman 1979] is the classical supporting tool. The user describes the files containing the modules and the dependencies between files in a *Makefile*. This information together with some implicit rules (which might be hard to understand) enable Make to re-create the executable code after a change has been made to the source code.

Make is language independent and general in that it does not only support compilation and linking – any user-specified commands can be executed on the files dependent on the ones that have been changed. The general rule is [Feldman 1979]:

> To "make" a particular node N, "make" all the nodes on which it depends. If any has been modified since N was last changed, or if N does not exist, update N.

Creating and maintaining Makefiles may be a cumbersome task; it is up to the user to continuously infer dependencies and ensure that the referenced files actually exist. Commonly used languages may have utilities for automatic generation of dependency descriptions. For example, *makedepend* [Brunhoff 1991] for C describes dependencies between a source program and its "#include" files. Some PC compilation systems keep track of changes and initiate necessary compilation and linking automatically without the need for any user-maintained "Makefile". The THINK C™ product [Symantec 1989] for the Macintosh, with its Auto-Make facility, is one example.

3.3.2.2 Smart Recompilation

In large application systems, recompilations represent a significant part of the maintenance costs and may thus be a hindrance for required system evolution. Avoiding unnecessary recompilations is therefore an important issue. It has been reported that in a larger Ada application more than half of the compilations were redundant [Adams *et al.* 1989]. Make is not particularly helpful in avoiding unnecessary recompilations; it is unlikely that any language independent tool can be smart in that respect.

If a file containing declarations is shared by many programs, any change to that file typically initiates recompilations of all the programs – whether or not they use a changed declaration. Tichy [Tichy 1986] has proposed a "smart recompilation" method for reducing the number of recompilations after a change to such declarations. The compiler's[1] symbol table was extended to keep track of finer granularity dependencies between declarations (type definitions, constants, variables, macros, etc.) in a compilation context and the items in the compilation units referencing the declarations. The possible changes in the context are classified. For each kind of change, the dependency information is used in a test to decide whether recompilation is necessary.

An extension of Tichy's "smart recompilation" to "smarter recompilation" is described in [Schwanke and Kaiser 1988]. It is argued that Tichy's definition of compilation consistency could be relaxed without the risk of introducing new errors and thus reduce the turn-around time even further.

A proposal for reducing unnecessary recompilations by analysing the source code, detecting dependencies and then clustering related declarations, files, etc. is described in [Schwanke and Platoff 1989].

Also the linking time may be significant in large systems. An incremental linker that processes only the changed modules is reported in [Quong and Linton 1991]. The linking time is proportional to the size of a change, rather than to the size of the program. Use-dependency graphs are used in the implementation.

3.3.3 Other Configuration Management Tools

Many configuration management tools integrate the features of SCCS/RCS and Make and provide additional functionality. One example is DSEE [Leblang *et al.* 1985], which

[1] A Pascal compiler was used in a prototype implementation, but the method is generally applicable.

introduces the notion of *configuration thread* – a rule-based language for describing which versions of the source files that should be used to build the application system. Configuration management may also be incorporated in larger support environments such as Sun's NSE™ [Sun Microsystems 1988b]. Other major vendors like IBM, DEC and ICL have their own configuration management tools of various kinds. One recent example is the Vesta system at DEC [Levin *et al.* 1992] which is tailored for large-scale configuration management and includes a repository of Vesta objects and a (functional) programming language for describing configurations.

Several tools have also been developed by minor vendors and academic institutions; a representative selection of 15 products is described in [Dart 1991].

3.4 Tools Based on Static Program Analysis

This section describes tools supporting software construction and maintenance based on static program analysis. Such tools range from simple compiler enhancements to tools centred around internal repositories with sophisticated user interfaces.

3.4.1 Compiler Supporters

Some program analysis tools have been developed to compensate for the poor static checking by compilers for languages like COBOL, FORTRAN and C. For example, DAVE [Osterweil and Fosdick 1976] is an old tool developed for FORTRAN. The FORTRAN Toolkit [Parsys 1993] is a recent, more sophisticated example. LINT [Ritchie *et al.* 1978] is the classical tool for C. Such tools check, for example, that all variables are declared and that the number and type of actual parameters match the formal parameters. They may also perform checks for unreachable code, unused identifiers, same name for "different" objects, etc. which are not commonly found in compilers even for strongly, statically typed languages.

3.4.2 Data Flow Analysis

The tools described above are typically based on data flow analysis which is an analysis of variable usage in some path through a program [Fosdick and Osterweil 1976]. In addition to assisting in detecting anomalies and errors, data flow analysis has traditionally been used for compiler optimisation, but it may also be useful to aspects of software maintenance such as understanding existing software, impact analysis and verifying software after changes have been made [Keables *et al.* 1988].

A particular technique based on data flow analysis, also useful for software maintenance, is *program slicing* [Weiser 1982]. A large program is broken down into smaller pieces containing a set of statements related by their data flow – statements without influence on a given variable are stripped from the program. A more sophisticated scheme for statically breaking down large software applications is reported in [Gopal *et al.* 1992]. The software is decomposable along three dimensions (level, type and aspect), and various aspects of software maintenance can be supported by focusing on the appropriate decompositions.

3.4.3 Cross-Referencers

Simple cross-referencers generate annotated listings of the names used in a program and the types of the named objects. Each line declaring a name contains references to its uses, and each line where a name is used contains a reference to its declaration. Cross-referencers useful for large application systems have proprietary databases populated with source code information and provide command or query languages tailored for retrieving and analysing the static information. The databases are normally updated during compilation if a parameter is set. An example of a commercially available tool is SCA [DEC 1989] which is the source code analyser component of DEC's VAXset suit of CASE tools. FUSE [DEC 1993], running under OSF/Motif™, is another DEC tool that provides a sophisticated user interface including coloured call graphs (inconsistencies in red), different box types for different types of objects, pop-up windows containing the source text associated with icons on the screen, etc.

3.5 Meta-Databases

This section describes concept usage, standards and products in the field of meta-data management. A meta-database contains information about the definitions and uses of the data in an application system. The contents are thus meta-data (data about data).1 Information about the use of data is particularly useful for change management. Unfortunately, this kind of information is incomplete in most commercially available products. For example, the relationships between names in the schema, queries and programs are rarely tracked down. Nevertheless, the area is rapidly growing, and there are many proposals for sophisticated tools.

3.5.1 History of Development

The term *data dictionary* emerged in the late sixties [King 1967] and then denoted a collection of rather simple file and field descriptions. Typically, the information was similar to what can be derived from the data division of COBOL programs and was stored in a "computer-held data dictionary file" [King 1969].

1 From the viewpoint of this thesis, programs are data, i.e., information about programs is meta-data.

In most of the seventies the intention of a data dictionary was still to hold the names and definitions of data items used in an information processing system. In 1977, however, the Data Dictionary Systems Working Party of the British Computer Society [DDSWP 1977] extended the data dictionary concept further to include, amongst others, the description of the nature of each data item and cross-references to where they were used.

As time went on, the original data dictionary concept expanded its scope to denote a tool for storing information throughout the entire life cycle including analysis and design. This development was also reflected in the terminology. The term *repository* has been launched (cf. IBM's Repository Manager [IBM 1990]), and the concept is also referred to as *system encyclopædia.* The American National Standards Institute (ANSI) started work (in 1980) on a standard in which the term *Information Resource Dictionary System* (IRDS) was introduced [ANSI 1988]. ISO [ISO 1990] has also standardised a framework for IRDS (which does not agree with that of ANSI). A discussion on various data dictionary standards can be found in [Holloway 1988a].

Another variant of the concept, *data dictionary/directory*, has also been commonly used [Uhrowczik 1973, IBM 1980, Allen *et al.* 1982]. The dictionary component refers to textual and structural description of data elements, name usage, relationships between elements, etc.; the directory component refers to physical properties such as the location of the data, its internal representation, how it can be accessed, etc. However, as DBMSs have been introduced to achieve physical data independence, the information traditionally captured by the directory part has become less significant, which is also reflected in that the directory term is less used today.

3.5.2 Standards

The need for a common framework for describing data dictionaries or repositories resulted in a set of ISO IRDS standards [Spurr 1988]. A basic component of IRDS is the four-level model (see description in [Olle and Black 1988]). There are four levels and three level pairs, each of which consists of a higher level and a lower level. The higher level describes how information at the lower level can be represented (Figure 3.3).

- The Application Level is concerned with data relevant to end users of an information system (for example, a patient with name Smith is allocated to bed number 314).
- The IRD Level typically contains database schemata, class definitions in object-oriented systems or type definitions in strongly typed programming languages. This layer may also describe which programs, screens, queries, etc. operate on which parts of the schema.
- The IRD Definition Level describes what kind of information can be held at the IRD Level. For example, it defines whether "record type", "object type", "program",

"screen", etc. are concepts whose instances can be created at the IRD Level. The type system of a programming language would typically belong to the IRD Definition Level.

- The Fundamental Level describes the representation of information at the IRD Definition Level. For example, the IRDS Services Interface standard uses a special version of a SQL Data Definition Language. An entity-relationship model is another example.

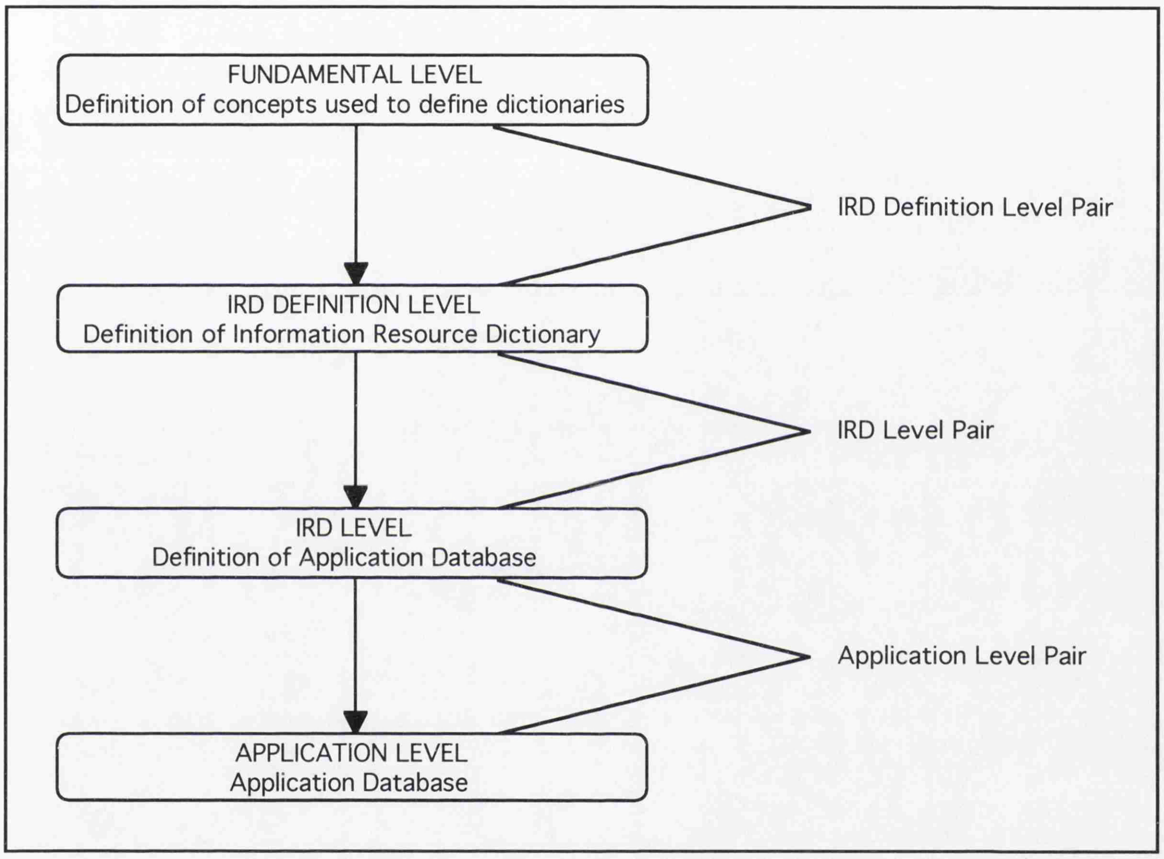

FUNDAMENTAL LEVEL
Definition of concepts used to define dictionaries
IRD Definition Level Pair
IRD DEFINITION LEVEL
Definition of Information Resource Dictionary
IRD Level Pair
IRD LEVEL
Definition of Application Database
Application Level Pair
APPLICATION LEVEL
Application Database

Figure 3.3: The IRDS levels and pairs

Since this model has become an ISO standard [ISO 1990], it is being used by leading tool vendors such as ICL [Kay 1992]. An example of a relational implementation of IRDS is described in [Dolk and Kirsch 1987].

The four-level model has also been adopted by the CASE Data Interchange Format (CDIF) standard [EIA 1991] developed by the Electrical Industries Association in the United States. CDIF is not a standard for repositories, but defines a standard that will enable repositories and CASE tools to interchange information in a standard format [Imber 1991].

Another related standard is Portable Common Tool Environment (PCTE) which is defined by the European Computer Manufacturers Association [ECMA 1990]. In addition to being a vehicle for data exchange, PCTE aims at becoming a coherent framework for integration of tools from various vendors running on various platforms. PCTE includes a repository (the Object Management System) and a set of services such as data and schema management, version and configuration management and inter-process communication.

A comparison of IRDS, CDIF and PCTE from the viewpoint of CASE data integration can be found in [Thompson 1992].

3.5.3 Features of Meta-Data Systems

Most data dictionary tools have been built for mainframes and thus have an old-fashioned user interface. Even though several products have versions for PCs or workstations, their interface capabilities are generally not fully exploited since the vendors strive for a common interface independent of platform. Another problem is the poor (if any at all) integration with other CASE tools such as configuration and build management tools. Rectifying these deficiencies are two of the desiderata of data dictionary users [Holloway 1988b]. Other desiderata, more relevant to this thesis, are automatic update, impact analysis and extensibility.

A crucial feature of meta-databases is to what extent they are automatically updated. Experience shows that manually updated information is rarely correct or up-to-date. This will in turn compromise tools that use the meta-data information as source for producing cross-reference information, call-graphs, methodology relevant information, etc. At least with the current technology, there is a trade-off between the spectrum of information contained in the meta-database and the degree of automatic update.

Information in meta-databases about software components and their relationships makes impact analysis possible. The quality of the analysis depends on the extent and granularity of the information. Many data dictionary tools do not support any automatic extraction of program information; others support COBOL or other "record-oriented" languages including so-called 4GLs. Some sophisticated products also support more modern programming languages, but the extracted program information is generally too coarse to produce satisfactory impact analysis.

It has been advocated that compilers and other static analysers should be tightly integrated with data dictionaries [Marti 1983]: "...the considerable overhead of populating a data dictionary during compilation is well worthwhile in computing environments with hundreds of data elements and a multitude of applications, where each comprises several thousands lines of code." Such an integration may enable both more detailed and more up-to-date information.

Extensibility is the ability to add user defined dictionary types at the IRD definition level (Figure 3.3). The user may require special types to support particular requirements which could range from detailed program information to support for a certain programming methodology, or even information relevant to requirements analysis and design. However, supporting extensibility, and at the same time automatic update, is a research issue of the future.

3.5.4 Commercially Available Products

This section describes briefly some commercially available meta-data systems in each of the categories *system catalogues*, *data dictionaries* and *repositories*.

3.5.4.1 System Catalogues

The (system) catalogue of relational DBMSs (DB2, INGRES, ORACLE, SQL/DS, etc.) is a kind of meta-database with a simple structure and a simple interface – it is a relational database like any other application database. The catalogue contains the database schema and additional information about indices, users, access privileges, etc.

3.5.4.2 Data Dictionaries

Software AG's Predict [SoftwareAG 1990] is a data dictionary system that includes query panels to the meta-data, cross-reference information, call-graphs (textual indention) and simple statistics on references (the number of times a certain field is referred to by catalogued program, for example). Predict is developed in and for a certain 4GL (Natural), but it also offers some support for COBOL, PL/1, FORTRAN and Ada.

Another sophisticated data dictionary system is ICL's DDS [Bourne 1979] which adheres to the IRDS architecture. There are a plethora of products from other vendors; some of them are compared in [Holloway 1988b].

3.5.4.3 Repositories

The repository concept – an extension of the data dictionary concept – emphasises the repository as a vehicle for tool integration. An example is IBM's AD/Cycle which provides "a framework for developing and maintaining applications throughout the entire development process" [IBM 1991]. AD/Cycle is a collection of application development (AD) tools and a platform providing services for the integration of these tools.[1] The Repository Manager [IBM 1990] is part of the AD/Cycle framework and provides an interface to a repository containing information about the data processing environment and other aspects about the enterprise's organisation, activities and processes. So, the repository contains information related to all phases of the application development life-cycle which is indeed a significant extension compared with what is currently held in system catalogues and data dictionaries. The idea is that such data can be further defined,

[1] Several components of the AD/Cycle concepts still have to be realised.

accessed, manipulated and controlled by tool builders and by other tools of AD/Cycle. User written tools and other software vendor's tools are also provided with these services if these tools comply with the standardised interface.

Other major software vendors have proposed sophisticated repository systems such as DEC's Cohesion, Hewlett Packard's Softbench and ICL's Open Dictionary.

3.6 Support Environments

Many attempts have been made to integrate supporting tools (such as those described in the previous sections) into software development environments. The variety of tools and environments makes classification difficult, but one may classify environments according to whether they are language independent or language specific, and whether they focus their support on the whole software life cycle or on the programming process. Table 3.1 shows the categories of some environments that will be referred to in the following sections.

	Language Independent	**Language Specific**
Whole Life Cycle	Eclipse (IPSE)	Arcs, Ada Env (APSE)
	VAXset, NSE	
Programming Process	Unix	Interlisp, Smalltalk env., Trellis env., Gandalf, Synthesizer Generator

Table 3.1: Categories of support environments

More detailed description and classification can be found in [Dart *et al.* 1987].

3.6.1 Language Independent Support Environments

The Unix programming environment [Dolotta *et al.* 1978], which is continuously being extended with new tools, is probably the most well-known language independent programming environment. DEC's VAXset and Sun's NSE are more sophisticated environments that support other life cycle phases as well.

An Integrated Project Support Environment (IPSE) aims at covering all the phases of the software life cycle. An IPSE should support in planning and control, provide office information facilities, be able to adapt to new technologies, etc. The idea behind an IPSE is that by integrating a large collection of software engineering tools into a common framework, the benefits should be greater than using the tools as separate units. Tools should communicate via an *object management system* (typically implemented on top of a DBMS), which is in contrast to integration around a file store (as in Unix).

Eclipse [Bott 1989] is one attempt at implementing an IPSE. The usefulness of IPSEs has yet to be demonstrated; there are indications that they control the process more than they support it [Sommerville 1993].

3.6.2 Language Specific Support Environments

Probably the most comprehensive language dependent programming environments are built, or proposed, around Ada, but many sophisticated programming environments have also been developed around other languages.

3.6.2.1 APSE

In connection with the development of Ada it was recognised early that in order to build large, complex and long-lived application systems, there was a need for a common support environment independent of the language processing environment. In the context of Ada this was called an Ada Programming Support Environment (APSE), and its requirements were defined in the Stoneman report [Buxton 1980].1 A description of the structure of an APSE can also be found in [Sommerville and Morrison 1987].

Arcs [Schefström 1991] is one example of a commercially available instance of an APSE. Arcs' vision [Schefström 1991] coincides with the motivation of the research presented in this thesis:

> The finding of a remedy for the frustration we feel when being confronted with a large piece of software that we want to modify, extend, or otherwise evolve, but cannot since we do not understand its structure and the possible effects of a change.

Rational's Ada Environment [Archer and Devlin 1986] is another example of an APSE.

3.6.2.2 Other Closed Environments

Interlisp [Teitelman and Masinter 1981] is a tightly integrated Lisp programming environment that contains tools such as Masterscope and DWIM (Do What I Mean). Masterscope analyses programs and stores cross-reference information in a database that can be queried. Masterscope can also invoke the editor on all functions satisfying the restrictions of a user query. A major component of DWIM is a spelling corrector. On the basis of contextual information DWIM can in several cases modify erroneous programs to contain the correct spelling. The variety of programming methodologies and tools accommodated has made Interlisp rather complex; even for highly motivated and skilled programmers mastery of all the tools has proved difficult.

The Trellis programming environment [O'Brien *et al.* 1987] supports object-oriented programming. Tools in the environment share a common database that is updated by an

1 In fact, the notion of APSE preceded the notion of IPSE (which is basically a language independent generalisation of APSE).

incremental compiler. The database contains source and object code, type checking information and cross-reference information. Another, well-known object-oriented programming environment is built around Smalltalk [Goldberg 1984].

Structure-oriented environments such as Gandalf [Habermann and Notkin 1986] and those generated by the Synthesizer Generator [Reps and Teitelbaum 1989] are another kind of language specific environments that are centred around a syntax-directed editor.

3.7 Summary

This chapter has presented models and tools intended to support various kinds of software evolution. Problems of schema evolution and build management have been discussed in particular, as well as tools such as static analysers, cross-referencers, data dictionaries (repositories), configuration management systems and support environments. Both research and commercially available products have been referenced.

The presented survey establishes a conceptual context for the research presented in Chapters 5, 6 and 7. Chapter 4 establishes the experimental context, which is persistent language processing technology.

Chapter 4

Enabling Technology

4.1 Persistent Programming

This chapter describes the pertinent features of Napier88 that is used for the experiments described in the subsequent chapters.

DBMSs have proved a useful basis for the organisation and management of large-scale, data-intensive application systems (e.g. the provision of physical data independence). The task of application programming, however, has become more complicated in that the programmer has to relate to a new set of concepts. The data definition, manipulation and query languages of the DBMSs support programming paradigms and data types that are normally incompatible with those of the traditional programming languages. A description of this problem was first published in [Atkinson 1978]. The problem has later been referred to as the "impedance mismatch" [Copeland and Maier 1984]. Figure 4.1 illustrates that the programmer must understand the set of mappings between each pair of the components: real world system, programming language and DBMS (data model).

Persistent programming languages were created to solve the problems mentioned above [Atkinson *et al.* 1982, Atkinson *et al.* 1983a, Atkinson *et al.* 1983b, Atkinson *et al.* 1983c]. Persistent programming unifies database programming and traditional application programming. This significantly simplifies the conceptual task of the programmer since only one mapping is needed – the one between the real world and the constructed system (Figure 4.2). The target of the persistent language designers was "to provide a totally integrated environment where the user never has to step outside the programming language for any computational activity" [Atkinson and Morrison 1985].

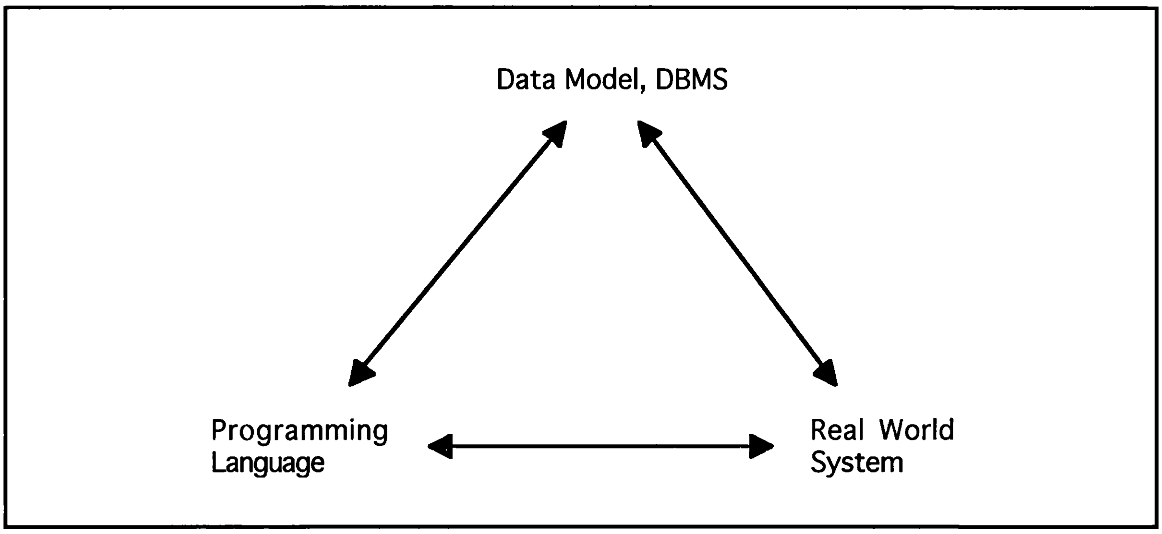


Figure 4.1: The three mappings of a traditional database system

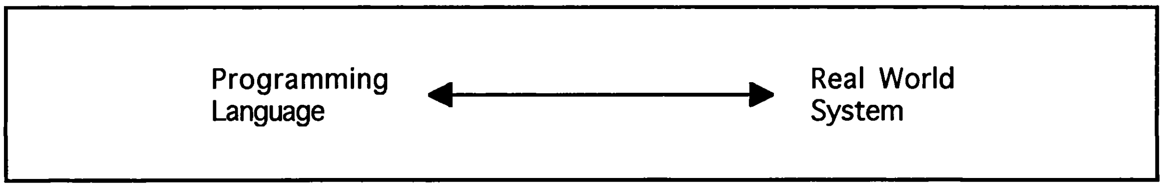


Figure 4.2: The only mapping of a persistent system

Two principles guide the provision of persistence:

i) *Persistence Independence*

   The semantics of a program is not changed by changes in the longevity of the data on which the program operates.

ii) *Persistence Orthogonality*

   The same facilities for persistence are accorded to data irrespective of the type of that data.

So, from the viewpoint of a programmer there is no boundary between data in the memory and data in the persistent store. He or she never has to write code in order to move or convert data between long and short term storage. In the persistent literature it is frequently quoted that typically 30% of all code is concerned with transferring data to and from secondary storage [IBM 1978]. In contrast, in persistent programming languages values of all types (a procedure, a complex tree structure, an instance of an abstract data type, etc.) have the same ability to outlive program executions.

One orthogonally persistent language [PS-algol 1987] has recently been successfully used in the implementation of *commercial* CASE tools [Greenwood *et al.* 1992]. *Research* in persistence is extensively reported [Atkinson and Buneman 1987, Atkinson *et al.* 1988, Dearle 1988, Atkinson 1989, Brown 1989, Cooper 1990a, Connor 1991, Atkinson 1992, Kirby 1993, Cutts 1993a].

4.2 Napier88

Napier88 [Morrison *et al.* 1989a] is a strongly typed orthogonally persistent language. It provides labelled Cartesian product (structures), labelled disjoint sums (variants) and explicit parametric polymorphism [Cardelli and Wegner 1985]. Existential polymorphism [Mitchell and Plotkin 1985] is used to implement abstract data types. Napier88 is a store-based language that combines persistence, higher-order procedures [Atkinson and Morrison 1985] and L-value and R-value binding [Morrison *et al.* 1990]. Persistence in Napier88 is defined by the model of reachability [Atkinson *et al.* 1983a], that is, an object will only outlive a program execution if it is reachable from one or more persistent roots.

4.2.1 Types

The Napier88 type system [Morrison *et al.* 1989b, Dearle *et al.* 1989] is based on the notion of types as sets of objects from the value space [Cardelli and Wegner 1985]. These sets are either built-in base types such as *integer*, *real* and *string*, or they are constructed by the use of built-in type constructors such as *structure* and *proc*.

A type expression may be given a name by declaring a type definition of the form as follows:[1]

```
[rec] type <identifier> is <type expression>
```

For example, the definition:

```
type Person is structure( name, address : string )
```

introduces *Person* as a type identifier that can be used later on in the program. Since Napier88 provides structural type equivalence, any type that is a structure of two string fields named *name* and *address* (either a type definition with another name or a type specified anonymously) is equivalent to the type *Person*. This may complicate certain issues such as determining whether a value conceptually belongs to *Person* or another type that happens to have the same structure. A discussion of type equivalence models in the light of persistent programming can be found elsewhere [Atkinson *et al.* 1988, Connor 1991].

[1] The syntax descriptions in this thesis are generally not complete; consult the manual for a full grammar description [Morrison *et al.* 1989a].

4.2.1.1 Type Databases

The Napier88 system provides a mechanism for storing pre-compiled type definitions in a database analogously to the meta-database used with conventional databases to hold schemata. Programs can be compiled against such a *type database*. Several type databases may exist within the same PAS – typically one for each sub-application. A type database is created or updated by compiling a program that consists only of type definitions. (The compiler must be invoked with a special command.) There are several advantages of type databases:

- they reduce the need to recompile type definitions;
- they enable sharing of type definitions; and
- they remove the need to duplicate type definitions and thus reduce the verbosity of programs.

Most Napier88 installations store the type representations in a PS-algol database, but the latest Napier88-in-Napier88 compiler [Cutts 1993a] uses Napier88 environments. It should be emphasised that the provision of type environments is only a convenience in the compilation process. The types in type environments cannot be accessed and bound to identifiers from within the language like other bindings. It is still more than an include facility as found in other programming languages, however, since the types are already compiled and interconnected with other types in the type graph that they use.

4.2.2 Higher-Order Procedures

In Napier88, as in for example ML [Milner 1984] and Quest [Cardelli 1989b], procedures are first class values. That is, they have the same civil rights as any other values in the language to be bound to identifiers, be assigned, be parameters of or returned by procedures, be elements of structures, variants or vectors, etc. It has been demonstrated that the combination of first class procedures and orthogonal persistence enables implementation of abstract data types, modules, separate compilation, views and data protection [Atkinson and Morrison 1985]. This powerful combination was exploited in the implementation of the tools described in the subsequent chapters.

Procedures that can take or return other procedures are referred to as *higher-order procedures*. In particular, the ability to return procedures may complicate error diagnostics and the provision of adequate program information based on static analysis. It may be difficult to determine which procedure activation returned a given procedure. The static universe of identifiers is different from the dynamic universe of identifiers. Essential to the understanding of these issues is the notion of *procedure closure* [Strachey 1967]. The following description is from [Atkinson and Morrison 1985]:

> The closure of a procedure includes all the information required to execute the procedure correctly. It has two parts. The first part is the code to execute the procedure, and the second part is its environment, which contains the local and free variables of the procedure and is often implemented by a static chain.

A mechanism for inspecting the closure – the source code and the state bound to it, i.e., the values of the local and free variables in the closure's environment – would provide the user with useful dynamic information. The hyper-programming environment [Kirby 1993] features such a mechanism (Section 8.2.6). There is a potential problem, however, in that allowing closure inspection will compromise the encapsulation of the procedure state (which is essential to the implementation of abstract data types, modules, etc. mentioned above).

4.2.3 Environments

An environment in a block-structured language is defined as the set of identifiers currently in a scope. The binding is static, that is, the set can be determined from the source code alone. Napier88 environments, which are of type *env*, are a dynamic model of the block-structured environments [Dearle 1988]. Environments are extensible collections of bindings, represented as name-type-value-constancy quadruples [Morrison *et al.* 1990], used to organise the persistent store. Moreover, environments are themselves first class values allowing the construction of arbitrary graphs with a root environment yielded by the built-in procedure *PS*.[1]

The standard procedure *environment* returns a new, empty environment. In addition, there are operations for inserting a binding into an environment, using a binding of an environment in a program, removing a binding from an environment and checking whether a binding is contained in an environment.

insert-declaration:	**in** <environment-clause> **let** <object_init>
use-clause:	$\underbrace{\textbf{use}\ \text{<environment-clause>}\ \textbf{with}\ \text{<signature>}\ \textbf{in}}_{\textit{header}}\underbrace{\text{<clause>}}_{\textit{body}}$
drop-clause:	**drop** <identifier> **from** <environment-clause>
contains-check:	<environment-clause> **contains** <identifier> [: <type_id>]

Figure 4.3: Operations on environments

[1] Most applications organise their persistent store as a hierarchy of environments.

A simplified syntax of the respective operations is shown in Figure 4.3. The leading phrase of each line and *use-clause header* and *body* are terms that will be used in the subsequent text.

4.2.3.1 Type Checking and Binding

The earlier in the software life cycle errors are detected, the less are the costs of correcting them. In particular, detecting errors during compilation (statically) is preferable to detecting them during execution (dynamically). The safest approach to avoiding run-time errors is to perform type checking and binding statically. There is a trade-off between safety and flexibility, however. From a persistent programming point of view there are several cases in which it is impractical to perform static binding [Atkinson *et al.* 1988]:

(i) reuse and distribution of programs;
(ii) selection of databases;
(iii) combination of information from several databases;
(iv) incremental data and program definition.

Hence, persistent languages like Napier88 have been designed with elements of dynamic binding but without compromising the quality and strictness of the type checking. The languages adhere to the principle of checking as much as possible as early as practical – for which Atkinson and Morrison coined the phrase *eager checking* [Atkinson and Morrison 1986].

In Napier88 the signature of the use-clause header (Figure 4.3) provides the point of dynamic type checking for incremental binding. In the use-clause body the type checking of identifiers declared in the signature is static. It is only necessary to specify the bindings that will actually be used in the program. That is, the signature needs only partially match the bindings in the environment. Therefore, the environments can be extended with new bindings without changing the existing programs. Bindings can also be removed safely as long as they do not occur in any use-clause. If during execution, however, a program attempts to access a binding not present in the respective environment, the following error message will occur: "Cannot find ⟨binding⟩ with type: ⟨type expression⟩."

Insert and drop are the two other environment operations that may cause run-time errors. If an attempt is made to insert a binding into an environment that already contains a binding with the same name, the following error message is given: "Attempt to re-declare ⟨binding⟩ with type: ⟨type expression⟩." An attempt to drop a binding not present in the environment results in the following error message: "Cannot drop ⟨binding⟩ it is not present."

A binding to a location is referred to as an L-value binding. A binding to the value contained in a location is referred to as an R-value binding. The reason for this naming is that in a block-structured languages an expression on the *left* hand side of an assignment operator is usually evaluated to the location, and an expression on the *right* hand side or any expression occurring elsewhere is evaluated to the value. In addition to this conventional left hand side evaluation, the binding to an environment expression in a use-clause header is also an L-value binding in Napier88.

There are no other languages with an explicit, dynamic environment construct like the one provided by Napier88. In a language with a static environment construct, e.g. Galileo [Albano *et al.* 1985], a program binds and type checks its environments completely at compile-time. Such a model is less flexible with respect to separate development, incremental update and dynamic choice of databases. This may be particularly impractical for large application systems [Atkinson and Buneman 1987, Morrison *et al.* 1990].

4.2.3.2 Separate Compilation

There are several reasons for organising separate compilation [Atkinson 1993]:

(i) to allow several people to construct parts of a system independently;
(ii) to allow parts of the system under construction or maintenance to be replaced;
(iii) to allow the construction and reuse of common subsystems such as libraries;
(iv) to enable incremental system construction of the form that has proved beneficial with databases; and
(v) to economise on computational time compared with re-compiling and re-linking total systems.

In conventional systems the incremental assembly of meta-data and data are organised under one regime, and the incremental construction and introduction of programs are organised via different mechanisms. The incremental program construction mechanism then involves three parts:

i) a means of identifying the component's context (schema name and procedures or abstract data types imported from other program parts);
ii) separate source text translation to some intermediate form; and
iii) linking all of the intermediate form fragments into one whole program (resolving and replacing the use of external names imported from other compilation fragments).

For safety in a strongly typed system the linking phase should verify that the names are being used correctly with respect to the type rules. This is a complex task, especially in the many systems that have general purpose linkers used by a variety of languages. Hence, it is often not performed or performed incompletely. Although this linking is performed statically[1] in conventional systems, much of the binding to schemata is resolved and verified dynamically as the program executes.

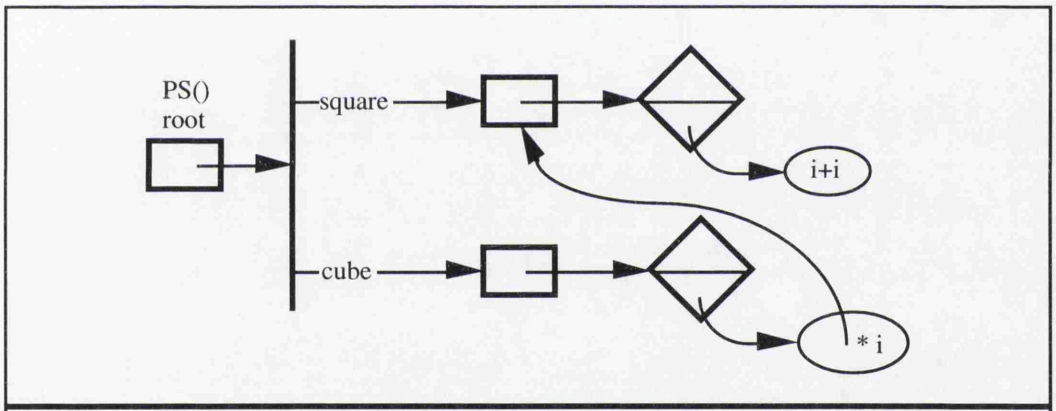
In contrast, Napier88 supports incremental construction that may be incrementally bound, is always fully type checked and uses a uniform model for all aspects of binding. It depends on the use-clause to identify the context and give enough information for complete

1 There are incremental linking systems that do it dynamically.

type checking prior to binding. It depends on environments to hold the existing program and data to which the new increment binds. The principle of data type completeness ensures that precisely the same mechanism and notations can be used for all the incremental binding requirements.

4.2.3.3 Some Napier88 Programs' Impact on the Persistent Store

The previous sections have stated that Napier88 facilitates incremental application construction by its environment construct, L-value semantics, separate compilation, etc. This section will illustrate how the state of part of the persistent store is changed by executing a few small, but complete, programs. The program examples and diagram technique are inspired by a lecture note in Napier88 programming [Atkinson 1993]. Other similar examples can be found elsewhere [Members 1990, Connor 1991]. First a naïve incremental method is shown. This has deficiencies when code is to be revised. An incremental method more accommodating to change is then presented.



```
in PS() let square = proc( i: int → int ); i + i          !Prog1.N – erroneous definition
```

```
use PS() with square: proc( int → int ) in                 !Prog2.N
in PS() let cube = proc( i: int → int ); square( i ) * i
```

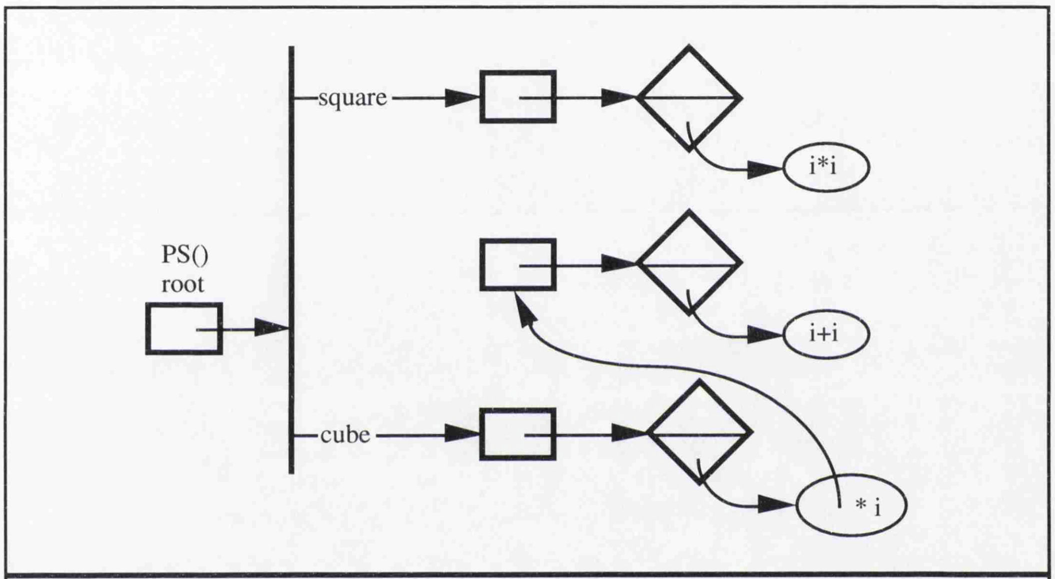
Figure 4.4: Part of the store after running Prog1.N and Prog2.N[1]

Figure 4.4 shows the store after the programs Prog1.N and Prog2.N have been compiled and executed. Prog1.N inserted the procedure *square* into the persistent store. (In

1 Solidly drawn boxes are locations that cannot be updated; solidly drawn vertical bars are values of type **env**; the identifiers to the right of them are identifiers in the bindings held in that environment; the lightly drawn boxes (appearing in Figures 4.7 and 4.8) are L-values (which can be updated), and the divided diamonds are procedure closures with the bottom part denoting the code and the top part denoting the computational context; the ellipses are code segments corresponding to procedure bodies; there may be arrows coming out of the ellipses corresponding to references to procedures or other boxed data that the procedure bodies use.

practice, such a procedure would be inserted into an appropriate environment at a lower level, but for simplicity in the following examples all bindings are inserted directly into the top level.) The procedure *cube* of Prog2.N used *square* in its definition, which is shown in the figure by the arrow from the ellipse body of *cube* to the location of *square*. Both *square* and *cube* are declared as constants which means that their locations are immutable.

We realise that the definition of *square* was wrong. Before we can insert a corrected version, the existing *square* must be dropped. This is performed by Prog3.N. After Prog3.N has been executed, the state of the store is as shown in Figure 4.5. Note that *cube* still uses the old version of *square*.

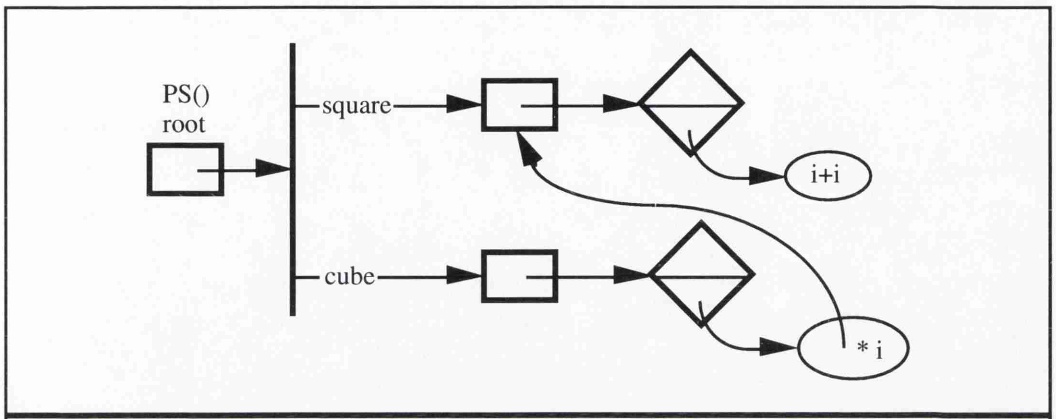


```
drop square from PS()                                  !Prog3.N
in PS() let square = proc( i: int → int ); i * i     !correct version
```

Figure 4.5: Part of the store after running Prog3.N

However, dropping *cube* and then re-defining it by referring to the new *square* (as shown in Prog4.N) will give a correct version. Figure 4.6 shows that after executing Prog4.N there is no reference to the old *square,* and thus it will be garbage collected.

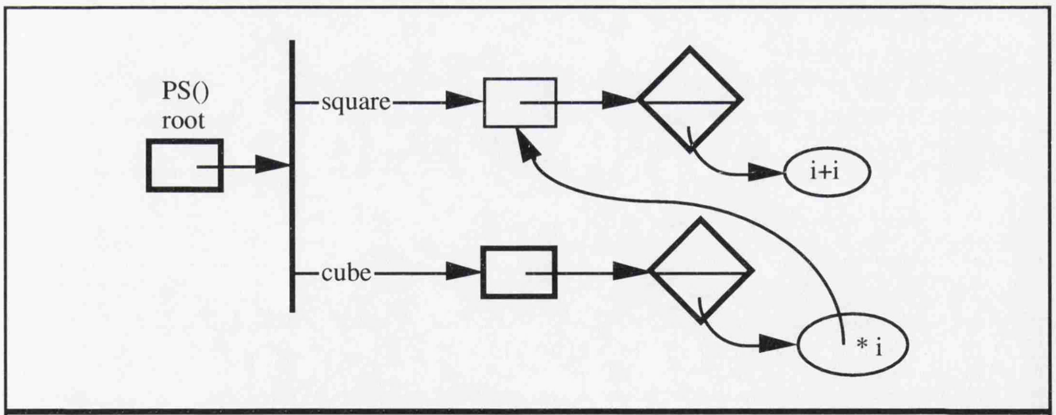
The strategy shown above requires ever increasing numbers of programs being dropped and redefined as cascades of "false" changes ensue from one correction. An incremental method of programming, which simplifies changing values, creates procedures as L-values.



```
use PS() with square: proc( int → int ) in                !Prog4.N
begin
      drop cube from PS()
      in PS() let cube = proc( i: int → int ); square( i ) * i
end
```

Figure 4.6: Part of the store after running Prog4.N

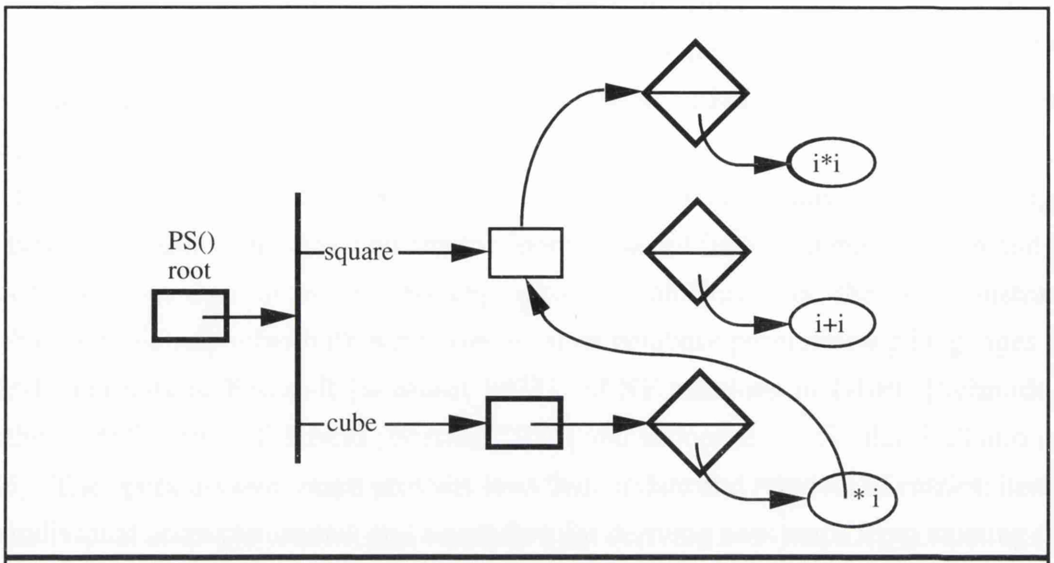
Prog5.N is equivalent to Prog1.N except that *square* is declared as a variable, rather than a constant, implying that its location is mutable. If Prog5.N and then Prog2.N were executed, the store would be as shown in Figure 4.7. The only difference from Figure 4.4 is that *square* is represented by a lightly drawn box.



```
in PS() let square := proc( i: int → int ); i + i          !Prog5.N – erroneous definition
```

Figure 4.7: Part of the store after running Prog4.N and Prog2.N

Since *square* is now bound to its L-value, the contents of the location can be changed by simply compiling and executing Prog6.N (Figure 4.8). Since *cube* is bound to *square's* location, the change will be directly visible to *cube* without the need for any recompilation or re-execution. The closure with body "i+i" has no references and will thus be garbage collected.



```
use PS() with square: proc( int → int ) in                    !Prog6.N
        square := proc( i: int → int ); i * i
```

Figure 4.8: Part of the store after running Prog6.N

The program examples shown are extremely small and simple. In large application systems it may be very hard to keep track of the structure of the persistent store as programs are being executed. Supporting methodologies and tools are obviously needed.

4.3 The Napier88 Programming Environment

A Napier88 release includes a standard environment containing commonly used procedures and other values. In addition, the Napier88 programming environment[1] includes a callable compiler [Cutts 1993a], the WIN window manager [Cutts *et al.* 1990], browsers [Kirby and Dearle 1990, Farkas *et al.* 1992], hyper-programming[2] features [Kirby 1993], both model and schema editors [Qin 1993] and the maps library (Section 4.3.1). Facilities for copying values between persistent stores have been implemented, and current research aims at providing Napier88 with concurrent technology for distributed systems [Munro 1993].

Napier88 has proved a robust and stable language platform both for teaching and research, and the collection of libraries is continuously being extended [Atkinson *et al.* 1993].

The current Napier88 compilers are running under Unix on Sun SPARCstations; each persistent store is contained in a (big) Unix file.

1 This environment should not be confused with the environment construct discussed above.

2 See Section 8.2.6.

4.3.1 The Maps Library

The maps library [Atkinson *et al.* 1990, Atkinson *et al.* 1991a, Atkinson *et al.* 1991b] is heavily used in the implementation of the tools described in the subsequent chapters. Maps constitute an add-on bulk type language implemented as a library of polymorphic Napier88 procedures.

Formally, maps are extensional functions from a domain of any type *A* to a range of any type *Z*. Values of this type constructor denote a stored finite mutable function and may be considered as a set of tuples. By appropriate parameterisation the map construct is capable of modelling other bulk types used in other database programming languages such as 1NF relations in Pascal-R [Schmidt 1977], NFNF relations in DBPL [Schmidt and Matthes 1992], sets in P-Pascal [Berman 1991] and sequences in Galileo [Albano *et al.* 1985]. The operators over maps provide: insertion, update and removal of entries; iteration and individual access to entries; and an algebra for deriving new maps from existing maps with a power similar to that of relational query languages or set comprehensions.

4.4 Napier88 Language Processing Technology

The Napier88-in-Napier88 (NinN) compiler is a procedure in the persistent store that can be called dynamically and facilitates a particular form of reflection [Maes 1987]. Programs that operate on the persistent store can be generated and executed at run-time which means that programs can change their own environment. This ability is referred to as run-time linguistic reflection [Stemple *et al.* 1992].

The NinN compiler is built according to the single-pass technique of recursive descent compiling [Davie and Morrison 1981]. The syntax analyser of productions in the grammar defining language is provided by corresponding recognition procedures. These procedures are mutually recursive in compliance with the mutually recursive definition of the language. Other procedures focus on lexical analysis, type checking, code generation and error handling. Still, the various components of the current compiler are rather intertwined making reuse of separate components difficult. Work is in progress, however, to identify substitutable generation interfaces for syntax analysis, code generation, etc. which will simplify reuse [Cutts 1993b].

Tools for browsing, hyper-programming, etc. need to determine the names and types of the bindings in the persistent store. The implementation of the Napier88 browser includes facilities for scanning the store [Kirby and Dearle 1990]. A first implementation used linguistic reflection, but owing to unsatisfactory performance the browser was re-implemented using low-level technology [Kirby 1993]. This technology is a collection of procedures that are not type-safe and are thus not available in standard Napier88; they are only accessible via a special system-builders' version of the compiler.

4.5 Summary

Napier88 is an orthogonally persistent programming language that provides a sophisticated type system, first class (polymorphic) procedures and an environment construct for organising programs and other data in the persistent store. The language is designed to facilitate the construction and maintenance of long-lived, data-intensive application systems. Among other things, application programs can be stored as values within the store and as such are susceptible to manipulation by change management software. Also, since several useful libraries exist for Napier88 (e.g. maps) and the language processing technology has proved robust and powerful, Napier88 was chosen as the experimentation and implementation language for the methodology and tool research described in the following chapters.

The persistent technology itself also provides some challenging problems for methodology and tool construction. Napier88 is still in its infancy as an implementation language for large-scale applications. For example, guidelines and tools are still needed to organise the interaction between programs and bindings in the persistent store.

Chapter 5

TSIT – A Thesaurus-Based Software Information Tool

5.1 Introduction

In order to fully benefit from the features of persistent programming, a programmer should be assisted by tools that help in keeping track of the structure of the programs and the persistent store. He or she may want the answers to questions like: Which environments, types, procedures, etc. exist? In which programs or environments are they defined or used and in which contexts do they occur? Which programs operate on which environments? The need for tools providing such information has often been experienced by persistent programmers, for example when the integrated Thesaurus Application was built – a multi-author, multi-level project [Sjøberg *et al.* 1993].

One proposal in this direction is the Thesaurus-based Software Information Tool (TSIT). In TSIT the ideas and principles behind the HMS thesaurus tool (Chapter 2) have been further developed. The information provided is different, however, since TSIT operates in a strongly typed, persistent programming environment, while the HMS thesaurus tool operates in an untyped, conventional programming environment.

For each application using TSIT there is an associated thesaurus holding the names bound to Napier88 concepts (type definitions in addition to environments, procedures and other values). The (meta) data in the thesaurus is generated by the analyser component of TSIT which scans all the source files and all environments in the persistent store of the actual application. Each time a name occurrence is encountered, the name and associated information are stored in the thesaurus. To ensure correctness and consistency, thesaurus entries cannot be inserted, modified or removed interactively or by any program that is not part of TSIT. This is not enforced, but no difficulties with inconsistency (due to programs that do not comply with TSIT's consistency expectations) have been encountered.

As will be demonstrated in this chapter, the thesaurus may form a basis for various kinds of measurements. The thesaurus also provides an appropriate platform for other software engineering tools (to be described in Chapters 6 and 7).

5.2 The Napier88 Thesaurus

The heart of TSIT is the thesaurus which is a fine-grained, enhanced cross-reference database containing information about all user-introduced names occurring in the source programs of an application and the names of the bindings in the associated persistent store. There is one thesaurus entry per identifier occurrence (declaration or use). The information held by a thesaurus entry is as follows:

- *Name* is a textual form of an identifier in a source program or of a name-type-value-constancy binding in a persistent store.
- *Container* indicates whether the entry is contained in an environment or in a file.
- *Block depth*, *block sequence* and *line number* of the identifier occurrences are meaningful and are recorded if the container is a file.
- *Kind* is an approximate representation of the type, i.e., base type (integer, real, string, etc.) or constructed type (structure, variant, (polymorphic) procedure, ADT, etc.).
- *Constancy* shows whether the identifier was declared constant or variable.
- *Usage* indicates how the identifier is being used, e.g. declaration or use of a type identifier, or declaration, left context or right context of a value identifier.
- *Context* indicates whether the identifier occurs in an environment operation or as a declaration of a type parameter, procedure parameter, structure field, variant tag, etc. or as a dereferenced structure field, projected variant, etc.
- *Date* keeps track of the date and time of when the entry was inserted.

Figure 5.1 shows the kind of data held about a thesaurus entry. Rectangular boxes represent structures; rounded boxes variants. The *container* field indicates whether the name denotes an identifier in a source program contained in a file or a binding in an environment in a persistent store. The directory path and file name are recorded if the container is a file. Similarly, the environment name and the path from the persistent root are recorded if the container is a persistent environment. (As opposed to files, environments can be organised in any structure – not only hierarchies.)

If the container is a file, the line number of the name occurrence is stored in the thesaurus together with block depth and block sequence which yield information about the scope of an identifier. Block depth is the number of nested 'begin's (or '{'s), i.e., the number of encountered 'begin's (or '{'s) minus the number of encountered 'end's (or '}'s). Block sequence is the total number of 'begin's (or '{'s) encountered before the name occurrence.

The *kind* field is a variant holding information about the *type* of the associated identifier if it has a base type or about the applied *type constructor* (structure, variant, proc, etc.) if it has a constructed type. In addition, the kind can be *Quantifier*, *TypeParameter*, *UnboundQuantifier*, etc. (Figure 5.1). A discussion of these concepts can be found elsewhere [Connor 1991]. (Note that *file* is a base type of a value that is not bound to a file in the file system but to an identifier in a Napier88 program. Such values should therefore not be confused with the files containing the Napier88 programs that constitute an application.)

The *usage* field informs whether the name occurs as a declaration or use of a type identifier, or as a declaration, left context or right context of a value identifier. For each of these five alternatives the name appears in a collection of variants that are referred to as *context*. For example, *typeDeclaration* is a variant having the context values[1] *RecursiveTypeDecl* and *TypeDecl* as tags, *typeUse* has the values *ArgUnaryOpType, ProcQuantifierUse, TypeNameUse, TypeParameterInTypeDecl* and *Witness* as tags, etc. The type of all these tags (and the tags of the *kind* variant) is *null* (omitted from the figure for simplicity).

[1] The tags of the variants defined in the type *Usage* are referred to as *context values.*

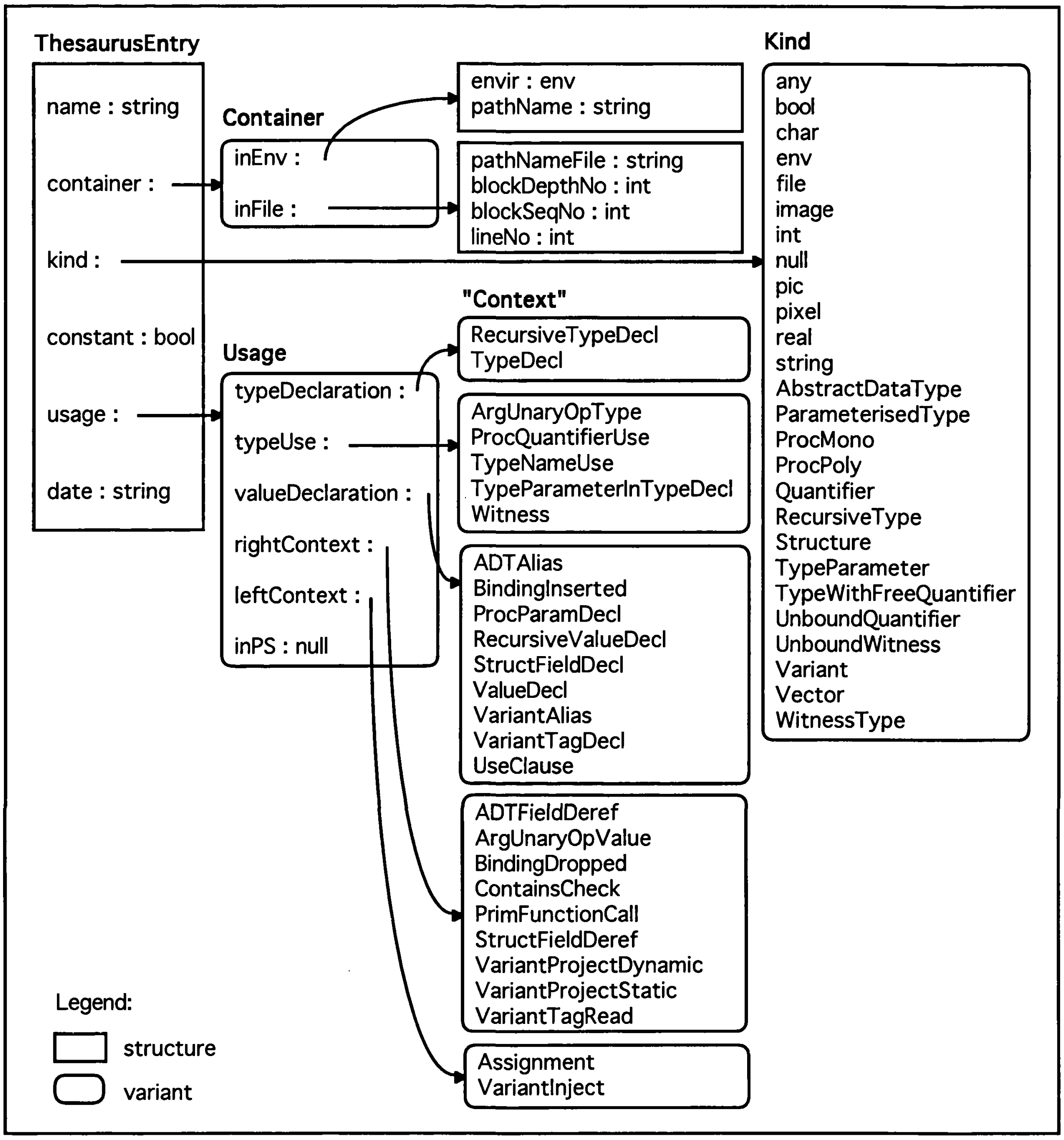


Figure 5.1: Definition of thesaurus entry

The contents of a thesaurus are best illustrated by an example. The corresponding thesaurus entries of the program *writePerson.N* (Figure 5.2) are shown in Table 5.1. Since these entries are contained in a file (the *container* variant is *inFile*), they are recorded with a file name and line number.[1] The *constant* attribute is represented in the table with a 'C' (constant) for true and 'V' (variable) for false.

[1] The block depth, block sequence and date information have been omitted from the table for simplicity.

```
type Person is structure( name : string; salary : int )

use PS() with IO, Adm : env                              in
use IO with writeString : proc( string );
            writeInt    : proc( int )                    in

in Adm let writePerson := proc( p : Person )
                          begin
                             writeString( "'nName: " ++ p( name ) )
                             writeString( "'nSalary: " )
                             writeInt( p( salary ) )
                          end
```

Figure 5.2: The program writePerson.N

Name	Container	L-No	Kind	Constant	Usage	Context
Person	writePerson.N	1	Structure	V	typeDeclaration	TypeDecl
name	writePerson.N	1	string	V	valueDeclaration	StructFieldDecl
salary	writePerson.N	1	int	V	valueDeclaration	StructFieldDecl
PS	writePerson.N	3	ProcMono	C	rightContext	PrimFunctionCall
IO	writePerson.N	3	env	V	valueDeclaration	UseClause:PS
Adm	writePerson.N	3	env	V	valueDeclaration	UseClause:PS
IO	writePerson.N	4	env	V	rightContext	ArgUnaryOpValue
writeString	writePerson.N	4	ProcMono	V	valueDeclaration	UseClause:IO
writeInt	writePerson.N	5	ProcMono	V	valueDeclaration	UseClause:IO
Adm	writePerson.N	7	env	V	rightContext	ArgUnaryOpValue
writePerson	writePerson.N	7	ProcMono	V	valueDeclaration	BindingInserted:Adm
p	writePerson.N	7	Structure	V	valueDeclaration	ProcParamDecl
Person	writePerson.N	7	Structure	V	typeUse	TypeNameUse
writeString	writePerson.N	9	ProcMono	V	rightContext	ArgUnaryOpValue
p	writePerson.N	9	Structure	V	rightContext	ArgUnaryOpValue
name	writePerson.N	9	string	V	rightContext	StructFieldDeref
writeString	writePerson.N	10	ProcMono	V	rightContext	ArgUnaryOpValue
writeInt	writePerson.N	11	ProcMono	V	rightContext	ArgUnaryOpValue
p	writePerson.N	11	Structure	V	rightContext	ArgUnaryOpValue
salary	writePerson.N	11	int	V	rightContext	StructFieldDeref

Table 5.1: The corresponding thesaurus entries for the program writePerson.N

If an identifier occurs in one of the contexts *BindingInserted, BindingDropped, UseClause* or *ContainsCheck*, the name of the actual environment is also registered.[1] For pragmatic reasons, the whole environment path has not been included in the current implementation since the combination of environment name and binding name is always unique in the analysed applications (but may not necessarily be in the general case).[2]

1 The environment name is the one occurring in the <environment-clause> described in Figure 4.3. If the <environment-clause> is a procedure returning an environment (e.g. *PS*), then the procedure name is recorded.

2 Identification of environments is discussed in Sections 7.5.2 and 8.2.5.

5.3 Querying the Thesaurus

Programmers may wish to query the thesaurus for debugging support, help in understanding the structure of an application, etc. Moreover, it may be beneficial to determine the effect of changes before actually carrying them out. Such impact analysis may influence the change plans – the consequences of change could be so extensive that another solution might be sought. A few examples of queries that can be performed on the thesaurus are:

- Which type definitions, procedures, structures, environments, etc. exist?
- Where are they defined and used?
- In which contexts do they occur?
- Which procedures, structures, etc. does a given program or environment contain?
- Which persistent procedures are used in a given procedure?
- Which bindings are inserted into or are in a given environment?
- Which operations are performed on which environments?

For large PASs the output from the thesaurus queries may be overwhelming. Some filtering mechanisms are therefore provided. The user can choose to exclude identifiers exceeding a certain lexical depth, identifiers with length one (e.g. counting variables in *for* loops) and contents of standard environments.

As the interface to the thesaurus and the query facilities provided by TSIT are rather primitive, a need was felt for a more convenient window-based interface with enhanced query possibilities. Lopes developed the ShTh component [Lopes 1993] by using WIN [Cutts *et al.* 1990]. ShTh provides a graphical interface to one or more thesauri and includes a simple query language, a subset of a generalised relational algebra, for operations on the thesaurus. Menu-driven facilities help the user to query a thesaurus and visualise the result of query application. "Select", "Project" and "Sort" menus are used to build a query. From an "Actions" menu the user has options to *load*, *close*, *save*, *save as*, *delete*, *revert* and *run* queries; it also has *undo* and *quit*.

Complex queries (involving recursion), however, cannot be expressed in the standard TSIT interface or the ShTh interface. To meet this deficiency, another software component, the ringad comprehension query language, was constructed by Trinder [Trinder 1991].

Typical data-intensive applications often use a powerful, usually embedded, query language for three reasons. First, although interactive query languages, like those provided by TSIT and ShTh, may be used by naïve users, they lack the computational power to express some useful queries. A thesaurus query that requires a powerful query language is procedure explosion. An explosion discovers all of the procedures that are called by a given procedure, and all of the procedures they in turn call, and so forth. Another thesaurus query requiring power implodes a procedure to find all procedures that call it, and any procedure that calls the caller, and so forth.[1] Information about explode and implode hierarchies (also referred to as call and contain trees) is presented to the user in a rather primitive way in the current version (only a label describing the level and then textual information about the procedure). An enhanced interface could visualise the different call levels by textual indention, as provided by Predict [SoftwareAG 1990], for example. A more sophisticated interface could be similar to the one provided by FUSE [DEC 1993] in which procedures and calls are represented as boxes and arrows in a colour graphical, window-based environment.

Second, a powerful, non-interactive query language can be used for "canned queries" which are queries or reports that are run regularly end-of-day, end-of-month, etc. Such queries are stored, i.e., canned, primarily to avoid errors. Storing a query may also aid efficiency and ensure that the information is always provided in the same format.

Third, many thesaurus users will have a high degree of computing skill and be able to use a powerful query notation themselves to extract information of interest about their programs. Incidentally, for any naïve users, the utility queries could easily be packaged into a menu.

Ringad comprehensions are a general purpose query language. In particular they can be defined over several different bulk types, e.g. maps, lists, ordered sets and vectors in Napier88. Comprehension queries are both powerful and easily optimised [Trinder 1991]. Because the utility queries access thesauri, which are maps, they use procedures out of the map library.

The experience of integrating the ShTh and comprehension query components with TSIT, in particular how the project benefited from persistence, is fully described in [Sjøberg *et al.* 1993].

5.4 Registration and Update

The thesaurus information is generated by a scan of all the registered source files and the registered parts of the persistent store associated with the actual application system. Each time a name occurrence is encountered, the name and additional information are inserted into the thesaurus. TSIT must be informed of the name and location of all the source files belonging to the application system. At present, TSIT reads a user-created file that contains the names of the respective directory paths and files. The user can also request analysis of one program at a time by specifying the path and name of the corresponding

1 Note that because Napier88 procedures are first class values, the list of called/calling procedures may not be exact. The tool is nevertheless useful.

file via an interactive menu. A simple enhancement could be to leave the localisation of the files to TSIT if the application were structured according to a convention such as that given as part of a methodology (see Section 6.3.7). If all the '.N' files in a directory and its subdirectories were part of the application, then TSIT would only need the top level directory as user input.

The TSIT interface also provides a menu for registering and scanning environments in the persistent store. TSIT extracts information about all bindings in a registered environment and recursively traverses all its subenvironments.

The contents of a thesaurus reflect a state of the corresponding application system. In order to reflect the continuous evolution of such systems, the thesauri must be updated correspondingly. It is not possible to add, change or remove entries from the thesauri manually. To ensure correctness and consistency, the contents should rely exclusively on TSIT. There are two exceptions, however. First, the possibility of removing all the entries of a given file has proved convenient when files have been included by mistake. Second, so-called *derived-thesauri* can be created as a result of querying the automatically generated *master-thesaurus* (simply called thesaurus in this thesis) [Sjøberg *et al.* 1993].

It is crucial that the thesaurus is as up-to-date as practically possible. There are several strategies for when to initiate an update:

i) Automatic initiation at regular times, e.g. daily at 02:00

ii) Update on user request

iii) Update during compilation

iv) Update during program composition – on edit

The first two strategies are possible in the current implementation of TSIT. Among other tools that analyse source code – both experimental [Marti 1983] and commercial [SoftwareAG 1990] – update during compilation is common. Typically, an extended compiler has a parameter indicating whether or not the program information database should be updated. In general, this strategy gives the most up-to-date information – if the user remembers to set the parameter. One might argue that the compiler could always update the database. In practice, however, the performance would deteriorate, and since most compilations are due to various kinds of bug-fixing (e.g. correcting typing errors), programmers will probably not accept the extra performance penalty.

TSIT was built as a tool separate from the compiler since the performance penalty pertaining to the thesaurus update would make it inconvenient to do the update during compilation.[1] The reason for the poor performance is that for each identifier occurrence encountered, an entry is created and inserted into the thesaurus in the persistent store.

[1] The separation from the compiler also made the implementation simpler.

Store operations are expensive at present, but current work aims at improving the performance [Atkinson 1992].

Poor performance is also a significant reason not to update the thesaurus during program composition. Another problem is that the code should be tested before information is extracted and the thesaurus updated. The code should at least be without compilation errors (a requirement of TSIT). So, the code should be compiled before it is analysed.

5.5 Implementation

TSIT is implemented in Napier88. The component of TSIT that processes Napier88 source programs is based on the NinN compiler [Cutts 1993a]. The lexical and syntax analysers of the compiler have been adjusted to conform to special information needs of TSIT. Instead of generating executable code, the TSIT analyser extracts a variety of information during the analysis and inserts it into the thesaurus.

Reusing the lexical analyser was straightforward – as opposed to reusing the syntax analyser. The NinN compiler is one-pass, i.e., the parsing and code generation are intertwined which means that detecting all program parts concerned with code generation is difficult. The documentation and some structuring principles alleviated the problem but were not sufficient for easy modification of the software to the needs of TSIT. In spite of this problem, the gain of reusing the compiler components was significant – developing TSIT would have been very much harder without reusing the NinN compiler.

The code for extracting information from the persistent store into the thesaurus was implemented by directly reusing low-level procedures used in the implementation of the Napier88 browser. This proved easy due to good documentation [Kirby and Dearle 1990].

The maps library [Atkinson *et al.* 1990] is heavily used in the implementation of TSIT. In particular, the predefined map operations enabled rapid development.

The type definition *Thesaurus* in Figure 5.3 shows that a thesaurus is a structure containing five fields. The type *ThesaurusEntries* defines a map of thesaurus entries (the *entries* field) where the domain (key) is a system-generated sequence number. (The range has already been described, see Figure 5.1.) The next number to be used is stored in the *nextSeqNo* field. The field *registeredFiles* denotes a map containing the registered source files constituting an application. The domain of this map is a concatenation of the directory path and file name. The range, specified by the *FileEntry* structure type, contains information about when a registered program was last compiled and executed. Section 7.4 discusses the use of that information. The *registeredEnvs* field is a vector containing direct references to the registered environments in the persistent store. Information about the type databases (Section 4.2.1.1) used in the PAS is recorded in *typeEnvs*. Finally, the type *Thesauri* defines a map containing a collection of thesauri, each thesaurus indexed by a name.

```
type ThesaurusEntries is Map[ int, ThesaurusEntry ]

type Date is structure( day, month, year, time : string )

type FileEntry is structure( compiled, run : Date )

type FileEntries is Map[ string, FileEntry ]

type Thesaurus is structure( nextSeqNo        : int;
                             entries          : ThesaurusEntries;
                             registeredFiles  : FileEntries;
                             registeredEnvs   : *env;
                             typeEnvs         : *string )

type Thesauri is Map[ string, Thesaurus ]
```

Figure 5.3: Thesaurus definition

The TSIT analyser and the queries performed on the thesaurus may be slow for large applications. Optimisation has not been emphasised in the current version. The implementation of the TSIT analyser prioritises easy modification over efficiency. For example, in an optimised version a few procedure calls could be saved for each name processed. Furthermore, shorter response times for the queries could be provided by additional data structures such as indices over entry name, file name, etc. and direct references between definitions and use of identifiers.

5.6 TSIT versus other Tools

Like the HMS thesaurus tool, TSIT records information about all names and identifiers used in the implementation of the whole application system.[1] The two tools are different, however, since their environments are different. For example, the HMS thesaurus tool was developed in an industrial, relational database context where four (untyped) languages were being used. In contrast, the development of TSIT benefited from a persistent programming context in which all computation and data management are dealt with within the same (strongly typed) language – Napier88. Moreover, since Napier88 is more sophisticated than the HMS languages, the information provided by TSIT is accordingly more sophisticated than the information provided by the HMS tool. Some of the differences are summarised in Table 5.2.

[1] Comments are not regarded as part of the code.

Criterion	**HMS Thesaurus Tool**	**TSIT**
context	industry	research
programming environment	multi-language (Hippo, Display Language, query dictionary language, schema definition language)	one language (Napier88)
analysis	interpretation	compilation
type checking	untyped, dynamic	strong, static and dynamic (stronger notion of type, user defined types, etc.)
program containers	files	files and environments in the persistent store
implementation	Unix scripts (*awk, grep, sed,* etc.) and one C program	modified Napier88 compiler and tailored Napier88 programs

Table 5.2: The HMS thesaurus tool versus TSIT

It may be worthwhile to briefly compare TSIT with the Napier88 browser [Kirby and Dearle 1990]. Both tools support the user in understanding the structure of an application, but their functionality differs. The browser provides *ad hoc* information about the contents of the persistent store selected explicitly on each occasion. TSIT also provides such information, but it is collected automatically. The user queries the thesaurus rather than browsing the store directly. The TSIT information may not be completely up-to-date since the contents of the store may have changed since the last thesaurus update. However, the browser does not provide any information about the source programs in the file system. (Traditionally, tools like Unix *grep* and the search facilities of editors have been used to locate identifiers in source programs.) As has been described in detail, TSIT can be queried for such selected information. Moreover, the thesaurus can be the subject of many forms of analyses such as application measurements (Section 5.7) and automatic consistency checks (Chapter 6).

Modifying compilers to generate dependency information is not a new idea. For example, DBPLXref [Matthes *et al.* 1992] is a tool that provides cross-reference information for applications built in the database programming language DBPL [Schmidt and Matthes 1992]. DBPLXref is not as comprehensive as TSIT. For example, detailed information about the context of each identifier occurrence is not provided, and the DBPLXref designers did not regard information about all local identifiers as interesting. On the contrary, the intention of TSIT was that the thesaurus information should be complete. For the purpose of various analyses all name occurrences should be recorded.

The main difference between TSIT and commercial source code analysers [DEC 1989] and data dictionary tools [Bourne 1979, SoftwareAG 1990] is a consequence of the mismatch in the underlying technology between programming languages and file systems/DBMSs (Chapter 4). The source code analysers view source programs as closed units and do not record information about how the programs interact with a file system or DBMS. The data dictionary tools emphasise database schema and file definition information. Some data dictionary tools also store source code and cross-reference information, but do not record dependencies between names in source programs and names in the schema (e.g. dependencies between variables in a program written in a language with embedded SQL and a field of a relation). As experienced when building the HMS thesaurus tool (Chapter 2), recording such information is quite complicated in a traditional programming environment. In the context of a persistent programming language like Napier88, however, it is trivial (as demonstrated by TSIT).

Even though some TSIT information may be more detailed (scope levels, contexts of identifier occurrences, etc.) and more integrated (source program versus persistent store information), the commercial data dictionary and repository tools generally have more extensive information related to the whole software life cycle (information about users, activities, documents, etc.). TSIT could be extended in that direction, but a problem is that such information is inserted manually. This is in contrast to the principle behind TSIT that all information should be generated automatically.

5.7 Measuring Name and Identifier Usage – A TSIT Experiment

As a guidance to research in language design, methodologies and tools for application development, this section presents measurements showing how programmers use the constructs of a higher-order persistent language like Napier88 and how they organise their software. Programmers may also benefit from such measurements.

The thesaurus contents of eight applications written in Napier88 were analysed. The analysis focuses on the use of *names*. A name in this context denotes an *identifier* in a source program. (Names denoting bindings in the persistent stores were not analysed since the author did not have access to all the individual persistent stores.) In most cases an identifier is uniquely denoted by its name. The same name can, however, denote different identifiers if they appear in different scopes. In those cases there are more identifiers than names.[1] All words that are not keywords of the language represent *name occurrences*. A name occurrence is simply an occurrence of a name independent of which identifier it denotes. For example, there are one name, two identifiers and three name occurrences in the following program:

[1] Actually, 13% of the names in the analysed applications denote more than one identifier.

```
let counter := 0
begin
    let counter := 1
end
counter := 1
```

Below follow some examples of questions that may be of interest to those designing tools, compilers and languages:

- How many different names are used?
- How many name occurrences represent value declarations, left contexts and right contexts, respectively?
- What is the distribution of names with respect to kind (base types or constructed types)?
- How frequently are type definitions used?
- How frequently are procedures used?
- What is the proportion of constants versus variables?
- How much code is concerned with operations on persistent store?
- How many programs update the contents of an environment?
- How many environments are updated within one program?
- How many declared identifiers are not used in each program? – in each application?
- How many type definitions are not used in each program? – in each application?
- How many bindings are inserted into the persistent store but never used in the application?

Measurements answering such questions may be useful in several respects. Programmers may get an overview of their software (e.g. the number of inconsistencies) and thus learn more about their way of programming. Language designers may wish to know how a programming language is actually used by programmers. Is the use of language constructs as expected? For example, the Napier88 language designers might question why abstract data types are hardly used in the analysed applications.

The main purpose of the name analysis, however, is the provision of measurements that may support or inspire the development of methodologies and tools for maintenance of persistent application systems. Measurements of the consequences of various kinds of change are of particular interest. For example, if the type of a procedure is changed, how many places in the programs have to be changed? What if a type declaration is changed? Where and how is persistent data created and modified? Such dependency statistics yield knowledge about consequences of change to various parts of an application system and thereby also about the extent of necessary change propagation.

Source code information from the following eight applications written in Napier88 has been collected and analysed:

- *Benchmark:* Sun engineering database benchmark [Birnie 1991];
- *Bibliography*: automatic generation of references in a text [Acheampong 1993];
- *Comp/TSIT*: a combination of a modified version of a Napier88 compiler [Cutts 1993a] and TSIT specific programs;
- *EcoSystem*: a graphical interface to an ecological database [Barclay *et al.* 1992];
- *ImplADT*: two implementations of parameterised abstract data types [Tabkha 1993];
- *Map*: a language construct for bulk data types [Atkinson *et al.* 1991a];
- *PartsDB*: an implementation of the parts explosion problem [Tabkha 1991]; and
- *WIN*: a persistent window management system [Cutts *et al.* 1990].

Bibliography and EcoSystem are the only real-world applications and developed by programmers not part of the "Napier88 community" in the Universities of Glasgow and St Andrews. Eleven programmers contributed to the application collection. In total, 51328 lines of code with 84501 name occurrences in 367 programs were analysed.

The study presented here is concerned with static aspects only. Similar studies have been reported for other languages, e.g. FORTRAN [Knuth 1972], PL/1 [Elshoff 1976], APL [Saal and Weiss 1977] and Ada [Agresti and Evanco 1992]. Also programs written in persistent programming languages have been analysed by others, but only dynamic aspects related to performance have been measured [Loboz 1989, Bailey 1989].

5.7.1 Scale of Analysis

Some measurements describing the size of the applications will be presented in order to give an impression of the scale of the analysis. Traditionally, programmers and project managers describe the size of their applications in terms of lines of source code. A better measure for the size may be the number of occurrences of programmer-introduced names of various kinds. Table 5.3 shows the size of the analysed applications in terms of lines of code. The applications consist of between 4 and 156 programs which each contains on average (*Mean*) 139.9 lines of code. Table 5.4 describes the applications in terms of name occurrences, where *Mean* is the average number of name occurrences in the programs. The number of different names within a program varies from minimum 10 (Comp/TSIT) to maximum 1945 (WIN). The last column contains the number of names per line, which is a measure for compactness of code, showing that the programs of ImplADT, Map and PartsDB are about twice as dense as the programs of EcoSys.

Application	Programs	Mean	Min	Max	Std	Sum
Benchmark	29	85.7	29	319	69.0	2484
Bibliography	38	170.8	14	449	109.7	6490
Comp/TSIT	80	104.5	6	427	87.7	8356
EcoSys	24	161.1	16	479	111.5	3867
ImplADT	11	104.2	8	303	85.1	1146
Map	25	193.8	9	541	176.3	4844
PartsDB	4	198.3	140	269	56.9	793
WIN	156	154.2	16	927	147.6	24053
Total	367	139.9	6	927	128.6	51328

Table 5.3: Lines of code

Application	Programs	Mean	Min	Max	Sum	Name/line
Benchmark	29	118.3	26	1141	3431	1.4
Bibliography	38	285.2	12	856	10838	1.7
Comp/TSIT	80	182.9	10	1141	14626	1.8
EcoSys	24	175.5	12	685	4213	1.1
ImplADT	11	241.1	12	786	2652	2.3
Map	25	379.2	13	1141	9479	2.0
PartsDB	4	429.0	286	645	1716	2.2
WIN	156	240.7	12	1945	37546	1.6
Total	367	230.2	10	1945	84501	1.6

Table 5.4: Name occurrences

5.7.2 Name Frequencies

A frequency analysis was performed on all names used in the applications. For example, in the Map application there are 641 different names which have 9479 occurrences in total. The number of occurrences of a given name varies between 1 and 1219. Such name usage information may encourage people to be more conscious of their choice of names and thus make programs more readable and understandable.

The histogram of Figure 5.4 shows the frequency of the times a name is used in the whole application collection, i.e., the number of names occurring once, the number of names occurring twice, etc. It appears that most names have 2 or 4 occurrences (respectively 14.3% and 12.4%). Moreover, 10% of the names are used 30 or more times.

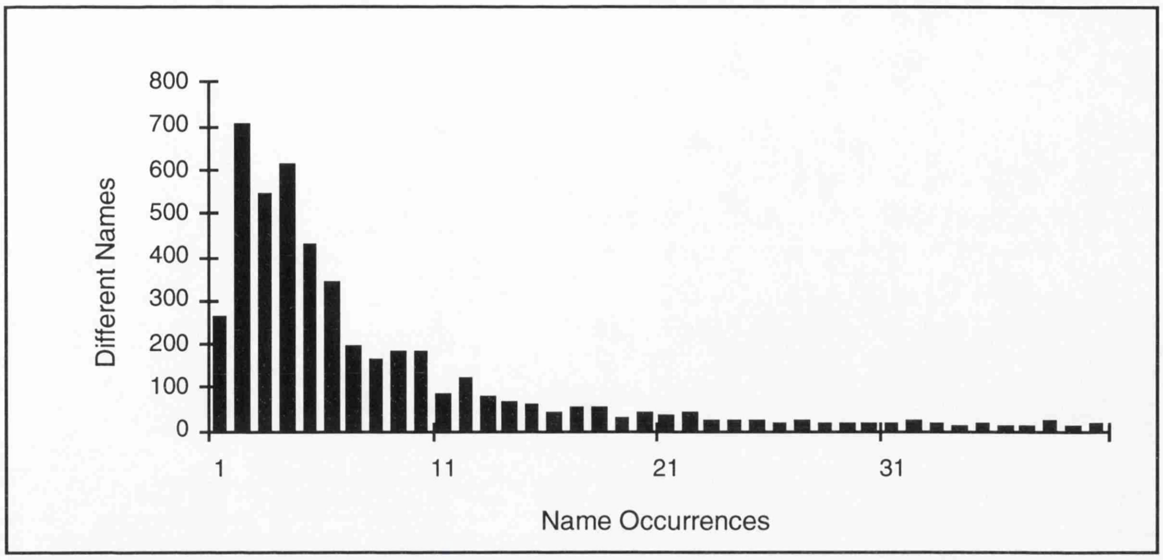


Figure 5.4: Name frequency

Name use within programs is described in Table 5.5. The *Names* column contains the number of unique (file name, name) combinations. *Mean* is the number of times the same name is used within a program on average and appears to be relatively stable in the applications (ranging from 3.3 to 5.5). The standard deviation, however, varies considerably (from 3.3 to 16.4).

Application	Names	Mean	Min	Max	Std	Sum
Benchmark	960	3.6	1	372	16.4	3431
Bibliography	3482	3.1	1	50	3.3	10838
Comp/TSIT	4406	3.3	1	372	9.1	14626
EcoSys	1338	3.1	1	68	4.1	4213
ImplADT	482	5.5	1	120	11.5	2652
Map	1879	5.1	1	372	15.1	9479
PartsDB	437	3.9	1	77	6.5	1716
WIN	9125	4.1	1	205	5.1	37546
Total	22109	3.8	1	372	9.5	84501

Table 5.5: Name use within programs

Table 5.6 shows to what extent names of identifiers are reused within programs, i.e., re-declared in another scope (re-declaration in the same scope is illegal). The table reveals that between 5.0% (Comp/TSIT) and 23.4% (ImplADT) of the names denote more than one identifier.[1] The applications Comp/TSIT, EcoSys, Map and WIN all have names that denote ten or more different identifiers within the same program.

1 The proportion of re-declarations is 100% minus the percentage of singular declarations (95.0% in Comp/TSIT and 76.6% in ImplADT).

Times Declared	Bench-mark	Biblio-graphy	Comp/ TSIT	EcoSys	Impl-ADT	Map	Parts-DB	WIN
1	91.0	92.7	95.0	88.0	76.6	79.4	84.5	82.2
2	6.8	6.1	3.2	9.8	13.2	12.1	10.1	8.5
3	0.6	0.8	0.9	1.2	5.7	4.1	3.3	4.3
4	1.5	0.2	0.4	0.3	2.3	2.2	0.9	2.7
5	0.1	0.1	0.1	0.1	1.8	1.2	0.9	1.4
6		0.0	0.1	0.2	0.2	0.5	0.2	0.4
7		0.1	0.1	0.1	0.2	0.0		0.1
8			0.1	0.1		0.2		0.0
9			0.0	0.0		0.1		0.0
>= 10			0.2	0.2		0.2		0.1

Table 5.6: Number of times a name is declared within a program (percentages)

5.7.3 Kind

The values are either of base types or constructed types. Constructed types are created by use of type constructors. A base type or type constructor is referred to as a *kind*. The infinite union of all types is *any*. Table 5.7 shows the distribution of the name occurrences with respect to kind. A question-mark indicates an unknown type. For example, when a binding is dropped from an environment, the type of the binding is not specified by the programmer.[1] It appears, among other things, that there are only 375 occurrences of abstract data types. Most occurrences are structures (19108).

The tendency of Table 5.7 is also reflected in the individual applications. Table B.1 in Appendix B shows the distribution of kind by application. Structure is the most frequent kind in six of the applications, and monomorphic procedure is either the most or second most frequent kind in another set of six applications. Some kinds vary significantly among the applications, however, such as unbound quantifier (from 0.1% to 29.1%) and polymorphic procedure (from 0.1% to 10.2%).

[1] The type information could have been extracted from the symbol table of the compiler but has not been regarded as important in the current analysis.

Kind	Freq	%	Kind	Freq	%
Structure	19108	22.6	TypeParameter	868	1.0
ProcMono	16299	19.3	RecursiveType	767	0.9
int	12231	14.5	null	731	0.9
env	10373	12.3	ParameterisedType	417	0.5
Variant	6029	7.1	ADT	375	0.4
UnboundQuantifier	5248	6.2	real	294	0.4
string	3866	4.6	file	248	0.3
ProcPoly	1790	2.1	UnboundWitness	84	0.1
image	1634	1.9	pixel	28	0.0
Vector	1617	1.9	pic	6	0.0
bool	1304	1.5			
any or ?	1184	1.4	Total	84501	100.0

Table 5.7: Distribution of kind

5.7.4 Name Usage and Context

The *usage* attribute defined in *ThesaurusEntry* (Figure 5.1) divides entries into type declarations, type uses, value declarations, left contexts and right contexts. Figure 5.5 shows how the entries are distributed among these options.[1] The pie chart reveals that declarations, left and right contexts of value identifiers constitute respectively 24.9%, 3.7% and 53.4% of all name occurrences. That is, a value identifier occurs in a right context about 2.1 times on average and in a left context about 0.2 times, indicating that identifiers are rarely updated compared with how often their values are accessed. Names of unbound quantifiers and type parameters constitute a large proportion (40%) of the type use. These type names are distinct from user-defined types in that they do not have any declaration – all their occurrences are classified as *TypeUse*.

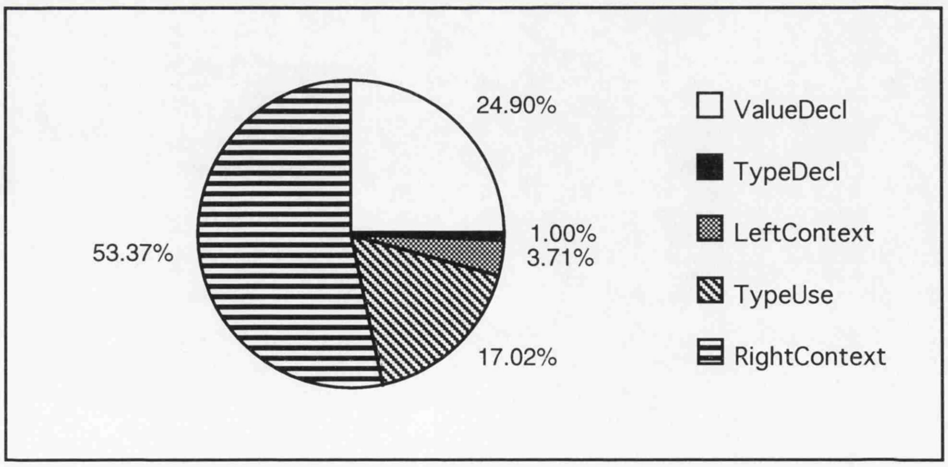


Figure 5.5: Name usage – total

1 All statistics on the whole application collection are weighted. For example, the weight of WIN (37564 thesaurus entries) is about four times the weight of Map (9479 entries).

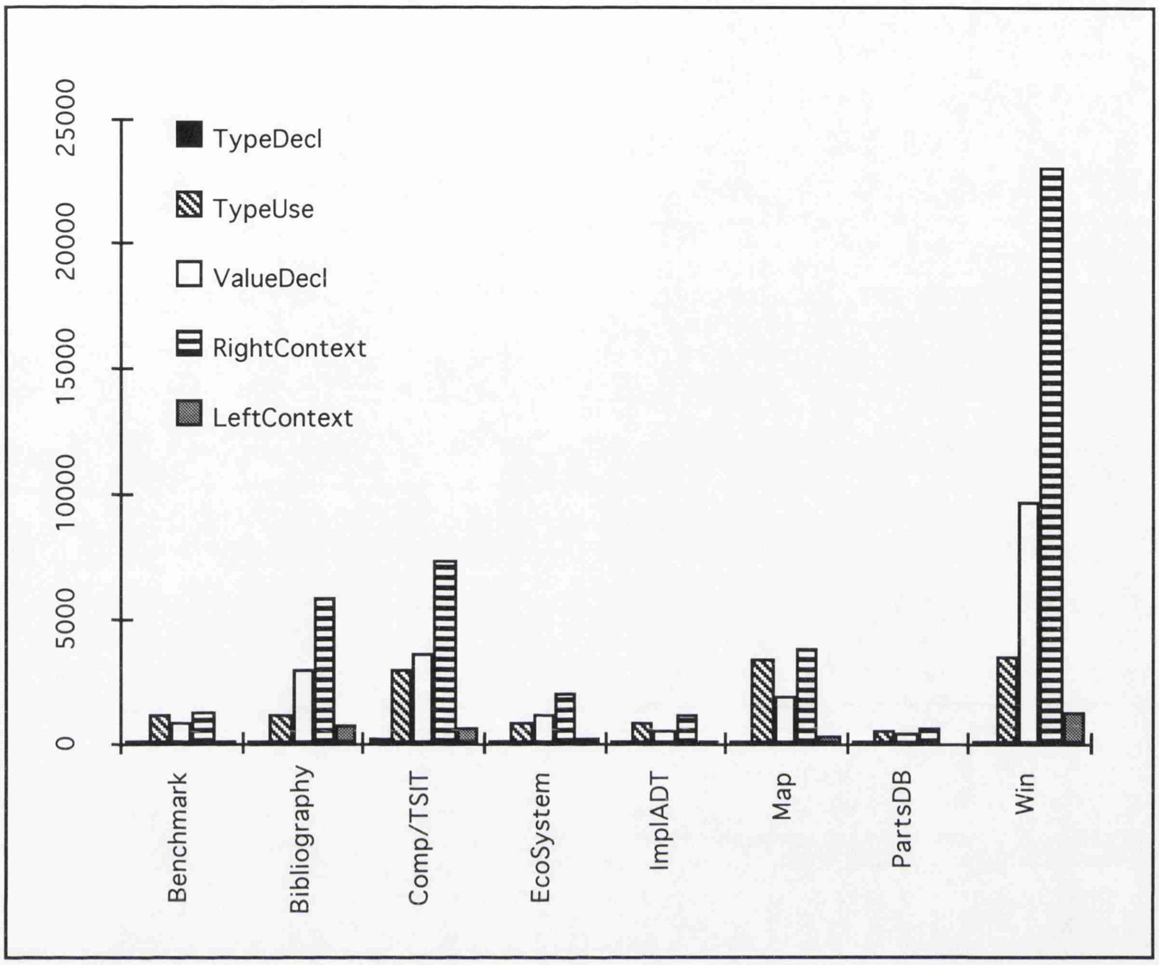


Figure 5.6: Name usage – by application

Figure 5.6 and Table 5.8 describe the distribution of usage for each application. *RightContext* has the largest value for all the applications. Thereafter follow respectively *ValueDecl* and *TypeUse* for half of the applications and vice versa for the other half.

Application	Type Decl	Type Use	Value Decl	R-Context	L-Context	Total
Benchmark	127	1179	798	1270	57	3431
Bibliography	122	1130	2973	5893	720	10838
Comp/TSIT	168	2951	3617	7312	578	14626
EcoSystem	50	818	1137	2039	169	4213
ImplADT	84	870	538	1101	59	2652
Map	116	3357	1899	3824	283	9479
PartsDB	109	537	420	643	7	1716
WIN	73	3541	9655	23015	1262	37546
Total	849	14383	21037	45097	3135	84501

Table 5.8: Name usage by application

The contexts are special cases of usages (Figure 5.1). Table 5.9 shows that the most frequent context is as an argument of a unary value operation, which includes: R-values in assignments, actual procedure parameters, use of environments, etc. Less frequent are, for example, contexts related to abstract data types (*ADTFieldDeref, ADTalias* and *Witness*). Differences and similarities between the applications with respect to context can be found in Table B.2 in Appendix B.

Context	Freq	%
ArgUnaryOpValue	36413	43.1
TypeNameUse	11124	13.2
UseClause	7618	9.0
ValueDecl	6726	8.0
StructFieldDeref	5338	6.3
ProcParamDecl	2911	3.4
Assignment	2503	3.0
ArgUnaryOpType	2074	2.5
StructFieldDecl	1693	2.0
BindingInserted	1584	1.9
VariantProjectDyn	956	1.1
ProcQuantifierUse	904	1.1
ContainsCheck	839	1.0
TypeDecl	684	0.9
VariantInject	632	0.7
BindingDropped	471	0.6
VariantTagRead	466	0.6
PrimFunctionCall	358	0.4
VariantTagDecl	325	0.4
PameterInTypeDecl	274	0.3
RecursiveTypeDecl	165	0.2
VariantProjectStatic	156	0.2
ADTFieldDeref	100	0.1
VariantAlias	86	0.1
RecursiveValueDecl	62	0.1
ADTalias	32	0.0
Witness	7	0.0
Total	84501	100.0

Table 5.9: Distribution of context

A "context by kind" table showing the distribution of kind for each context value (and a similar "kind by context" table) can be found in [Sjøberg 1992].

5.7.5 Constancy

In Napier88 a value identifier is declared as either a constant (the '=' assignment operator is used) or variable (the ':=' assignment operator is used). This section describes the distribution of constancy for the value identifiers.[1] Table 5.10 shows that 30% are constants and 70% are variables. There is hardly any difference between constants and variables with respect to how often they are used (the *rightContext* row). A constant is by definition not mutable and therefore cannot occur in a left context.

[1] Type identifiers are immutable and are thus not included in the constancy measurements.

Usage	**Constant**	**(%)**	**Variable**	**(%)**	**Total**	**(%)**
valueDeclaration	6288	(29.9)	14749	(70.1)	21037	(100.0)
rightContext	13392	(29.6)	31705	(70.3)	45097	(100.0)
leftContext	0	(0)	3135	(100.0)	3135	(100.0)
Total	19680	(28.4)	49589	(71.6)	69269	(100.0)

Table 5.10: Constancy distributed by usage

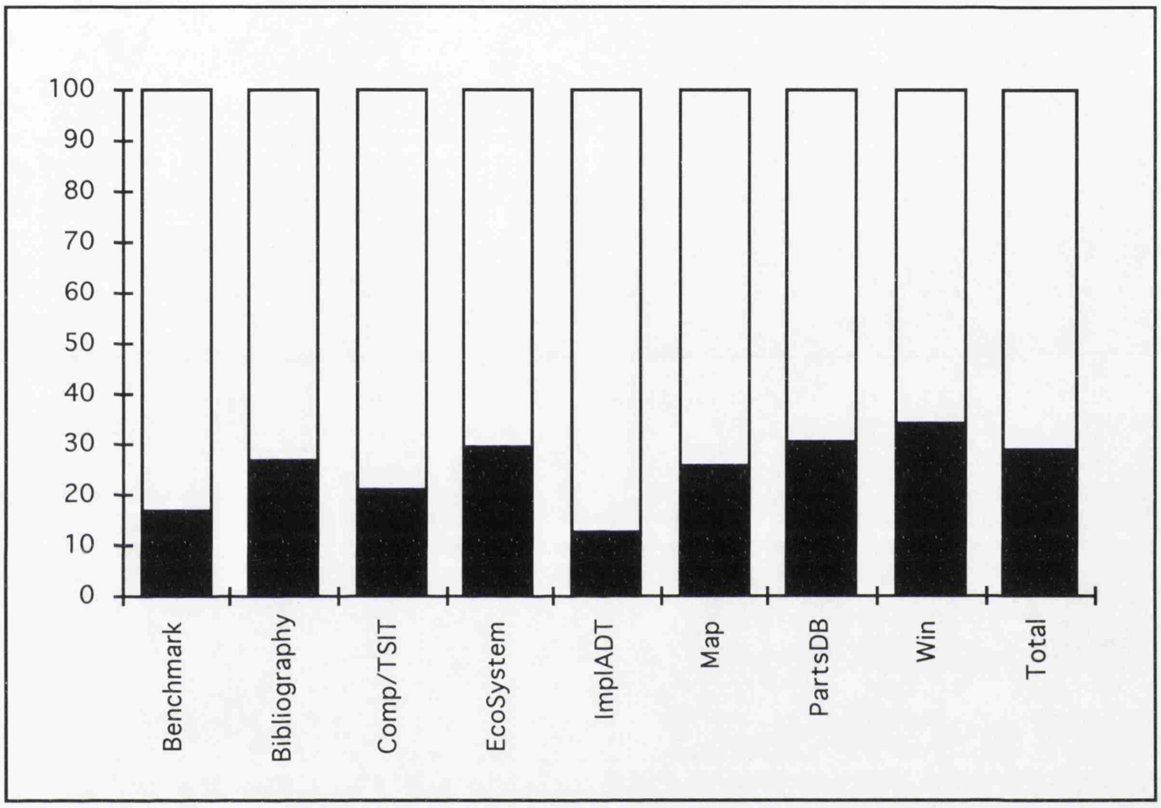


Figure 5.7: Proportion of constants in the applications

Constancy is distributed by application in Figure 5.7 revealing that the proportion of constants is relatively stable among the applications (perhaps with the exception of ImplADT).

It may be worthwhile to illustrate the constancy concept with respect to vectors since it may not be intuitive. In the program example of Figure 5.8 the first line declares a vector *v* as a variable and the second line an integer *i* as a constant. In line three both *v* and *i* appear in a right context[1] – implying that also *v* could have been declared as a constant. In that case, however, the assignment in line four would have failed. Table 5.11 contains the corresponding thesaurus entries of the program.

[1] So, even though the expression occurs on the left hand side of the assignment operator, the vector identified by *v* is not updated; the assignment applies only to the value of one of the vector's elements.

```
let v := vector 1 to 3 of "test1"
let i = 2
v( i ) := "test2"
v := vector 1 to 2 of "test3"
```

Figure 5.8: A vector program

Name	LineNo	Kind	Constant	Usage
v	1	Vector	V	valueDeclaration
i	2	int	C	valueDeclaration
v	3	Vector	V	rightContext
i	3	int	C	rightContext
v	4	Vector	V	leftContext

Table 5.11: Corresponding thesaurus entries

5.7.6 Name Length

The choice of names for identifiers is crucial for the readability of programs. One aspect of a name is its length. There may be different guidelines for the optimal length. Some examples follow:

i) Names should generally be long since long names can convey more information than short ones.

ii) The less frequently an identifier is used, the longer it should be.

iii) The greater the distance between identifiers, the longer they should be. (The distance could for example be measured in terms of number of lines or scope levels.)

iv) What is important is that the name is carefully chosen – which is independent of the name length (e.g. abbreviations can be very meaningful).

The appropriateness of these guidelines, which are not mutually exclusive, is not an issue of this thesis. The point is, however, that the thesaurus provides a means for testing the software against such guidelines. Only the distribution of the name length will be shown below. Identifiers denoting values in the standard environment are excluded since they would bias the result. Unbound quantifiers and type parameters have also been excluded since it is common practice to give them very short names (respectively 80% and 87% have length one). If two (or more) identifiers have the same name, that name has double (or more) weight. Figure 5.9 shows that most names have five or six characters. The average is 8.1 (Table 5.12). The maximum length is 29. (This information was useful when the screen interface of TSIT was implemented; the name length could be assumed not to exceed 30 characters.) Moreover, the table reveals that, for example, the ImplADT programmer has generally chosen names that are less than half the length of the Map names.

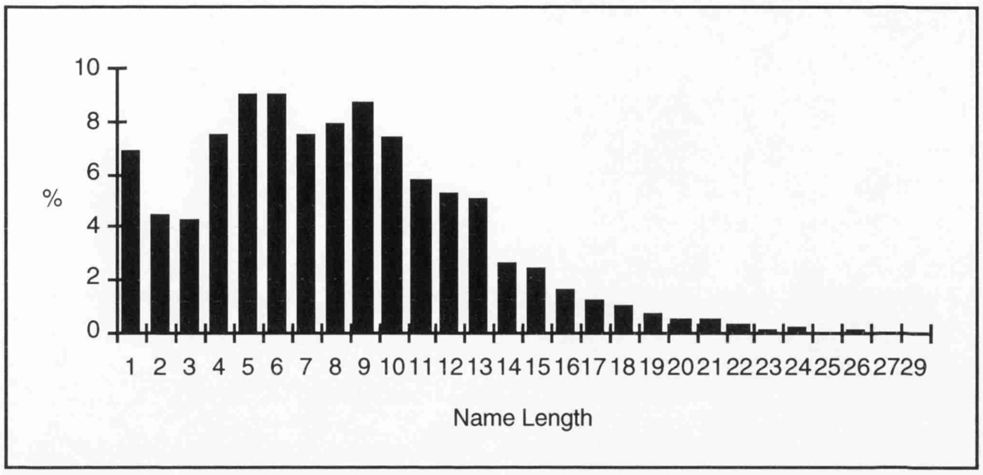


Figure 5.9: Distribution of name length

Application	Names	Mean	Min	Max	Std
Benchmark	748	8.3	1	26	4.8
Bibliography	2548	8.1	1	24	4.0
Comp/TSIT	3560	7.4	1	26	4.1
EcoSys	1062	8.9	1	21	3.9
ImplADT	598	4.2	1	14	3.1
Map	1812	9.3	1	26	5.6
PartsDB	518	4.8	1	12	2.7
WIN	8426	8.5	1	29	4.6
Total	19272	8.1	1	29	4.6

Table 5.12: Name length of type and value identifiers

An analysis of the name length distributed by kind showed that procedures have longer names than other kinds. There was no clear distinction between the other kinds.

5.7.7 Use of Type Definitions

The use of type definitions is illustrated in Figure 5.10.[1] A type definition used null times means that it is never used within the application. There are 208 (33%) such cases among the 626 different type definitions.[2] Moreover, 17% of all type definitions are used once, 6% twice, etc., and 10% are used 30 or more times with two extremes of 504 and 549 times.

1 A table containing the underlying numbers can be found in [Sjøberg 1992].

2 In this context a type name in one application is regarded as different from a type name in another application even though the name, and possibly the type expression, happen to be the same. However, the applications typically re-declare types defined in other applications such as libraries (Map and WIN). See also discussion in Section 6.3.2.

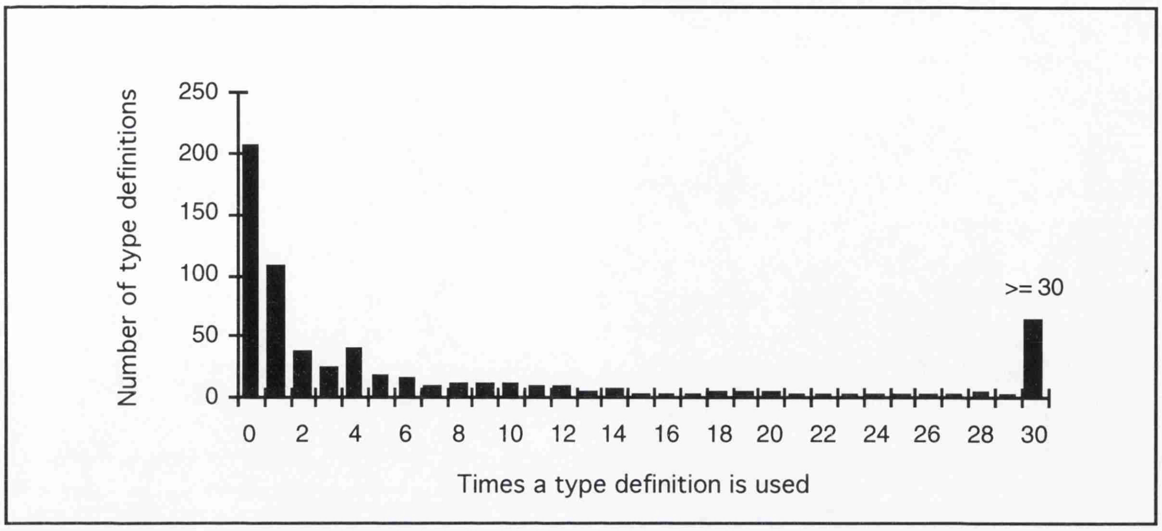


Figure 5.10: Distribution of use of type definitions

Application	Type Def	Mean	Min	Max	Sum	Std
Benchmark	125	3.0	0	87	370	8.5
Bibliography	85	8.5	0	74	722	10.9
Comp/TSIT	162	11.8	0	503	1904	47.5
EcoSys	40	18.6	0	177	742	33.0
ImplADT	15	12.4	3	65	186	11.2
Map	116	6.2	0	260	724	24.1
PartsDB	28	7.1	3	66	200	9.3
WIN	55	60.9	1	548	3349	99.5
Total	626	13.1	0	548	8197	40.9

Table 5.13: Statistics on the use of type definitions

Table 5.13 shows how many type definitions are defined in the various applications and some statistics on their use. The 626 type definitions are on average used 13.1 times. The average use varies significantly between the applications (from 3.0 in Benchmark to 60.9 in WIN). So, on average for all the applications a renaming of a type identifier will imply that 13 places must be edited. In the best case only the definition itself needs to be changed (*Min* is 0) while 548 places (*Max*) in the worst case. If the expression of a type definition is changed, the places of use must be changed depending on the context and whether or not the type is parameterised. Since type parameters and unbound quantifiers are not included in this section, there are basically two uses of type identifiers. First, the context may be *TypeNameUse*. That is, the type identifier appears in a declaration on the form <value identifier> : <type identifier> in the signature of a use-clause header or in a procedure parameter declaration.

Second, the context may be *ArgUnaryOpType*. That is, the type identifier is used to create instances of the type denoted as for example in:

```
let newPerson := Person( "Dag", "Glasgow" )
```

These two contexts constitute respectively 25% and 75% of the use of type definitions. In the former case a change to the type declaration does not affect the code if the type is not parameterised; only recompilation is necessary. If the type is parameterised, however, the place of use must be edited if the number of parameters is changed. In the current sample 26% of all type definitions are parameterised.

In the latter case a change to the type[1] denoted by an identifier must be propagated to all places where the identifier is used to create new instances. For example, if the type *Person* is extended with a field for occupation, that field must also be given a value as for example in:

```
let newPerson := Person( "Dag", "Glasgow", "Student" )
```

Table B.3 in Appendix B shows how many times a type definition is used in value instantiations. There are 166 different type definitions[2] which are used in 2074 instantiations implying that a change to the denoted type will affect 12.5 places on average. Moreover, 10% are used more than 26 times with three extremes of 130, 166 and 261 times.

Due to structural type equivalence in Napier88, the thesaurus information about the use of types may not be complete. That is, instead of the name of a type definition, anonymous types may be used in value instantiations and other declarations. For example, in the declaration:

```
let anotherPerson := struct( name = "Paul"; university = "Glasgow" )
```

the created value has the same type as the created value in the first instantiation of *Person* above. If the definition of *Person* is to be changed, then the declaration of *anotherPerson* should *probably* be changed as well. This illustrates that programmers may be encouraged to use named types in order to facilitate efficient change propagation.

[1] Being precise, in a language with structural type equivalence types are not changed. When saying that a type is changed, we mean that the type denoted by a type identifier has been replaced by another type. For example, in the first declaration of *newPerson* above the *Person* identifier denotes a type that is a structure with two string fields. In the next declaration *Person* denotes a type that is a structure with three string fields.

[2] Only 166 of the 626 type names are used in instantiations. The remainder are either used exclusively in the declarations of other types or not used at all.

5.7.7.1 Use of Structure Fields and Variant Tags

The use of structure fields and variant tags may be of special interest. If a field of a structure type is changed, all places where that field is dereferenced must be edited.[1] Table 5.14 shows the number of field declarations and dereferences. On average 3.2 places are affected, but the applications differ significantly (from 0.8 to 11.0). Table 5.15 is a similar table for variant tags.

Measurement	Bench-mark	Biblio-graphy	Comp/TSIT	EcoSys	Impl-ADT	Map	Parts-DB	WIN	Total
StructFieldDecl	67	440	259	196	175	41	174	341	1693
StructFieldDeref	79	527	915	252	244	450	141	2730	5338
Deref per field	1.2	1.2	3.5	1.3	1.4	11.0	0.8	8.0	3.2

Table 5.14: Use of structure fields

Measurement	Bench-mark	Biblio-graphy	Comp/TSIT	EcoSys	Impl-ADT	Map	Parts-DB	WIN	Total
VariantTagDecl	23	51	70	21	62	17	48	33	325
VariantDeref[2]	20	245	431	113	161	213	48	979	2210
Deref per tag	0.9	4.8	6.2	5.4	2.6	12.5	1.0	29.7	6.8

Table 5.15: Use of variant tags

A frequency table (context by kind) presented in [Sjøberg 1992] yields information about the distribution of kind for each context value. Two extractions, Tables 5.16 and 5.17, show respectively how many occurrences of structure fields and variant tags that are procedures, integers, recursive type declarations, etc. For example, most structure fields are monomorphic procedures or integers, whereas most variant tags are nulls or recursive type declarations.

1 This is always the case for deletion and renaming. For change to the type of the field or tag, however, there are cases where editing may be unnecessary (e.g. if the field or tag appears on the right hand side of an assignment).

2 *VariantDeref* is the union of the context values *VariantProjectDyn, VariantInject, VariantTagRead* and *VariantProjectStatic*.

Kind	Freq	%
ProcMono	547	32.3
int	326	19.3
RecursiveTypeDecl	179	10.6
Structure	169	10.0
Variant	156	9.2
string	133	7.9
Vector	69	4.1
bool	56	3.3
TypeParameter	23	1.4
real	15	0.9
image	10	0.6
null	4	0.2
ProcPoly	4	0.2
pixel	1	0.1
env	1	0.1
Total	1693	100.0

Table 5.16: Kind of structure fields

Kind	Freq	%
null	127	39.1
RecursiveTypeDecl	66	20.3
Structure	64	19.7
TypeParameter	39	12.0
string	12	3.7
image	5	1.5
int	4	1.2
Vector	2	0.6
real	1	0.3
pixel	1	0.3
pic	1	0.3
file	1	0.3
bool	1	0.3
Variant	1	0.3
Total	325	100.0

Table 5.17: Kind of variant tags

5.7.8 Use of Procedures

Code is contained in the persistent store in the form of procedures, which are therefore of particular interest. Their use frequency has been investigated. The histogram of Figure 5.11 shows that using a procedure only once is most common. There is also a relatively large number of procedures that are declared but never used (times used is 0).[1] The rightmost bar represents 30 or more uses. A table containing the exact numbers [Sjøberg 1992] shows that 10% are used more than 15 times including the maximum of 569 times (which is the *writeString* procedure). The average is 8.3; the standard deviation 28.9.

In these measurements a procedure is either monomorphic or polymorphic, and standard procedures are included.[2] Moreover, *procedure* here means *procedure name* and is independent of application. That is, even though two procedures with the same name are declared differently in different applications, programs or scopes, they are still counted as one procedure in this context. However, there are very few cases of such name duplication.

[1] Note that the samples include libraries; see also Section 6.3.3.

[2] A comparison between standard procedures and user-defined procedures showed that standard procedures were used more frequently, otherwise the two groups followed the same pattern (the results are not presented here).

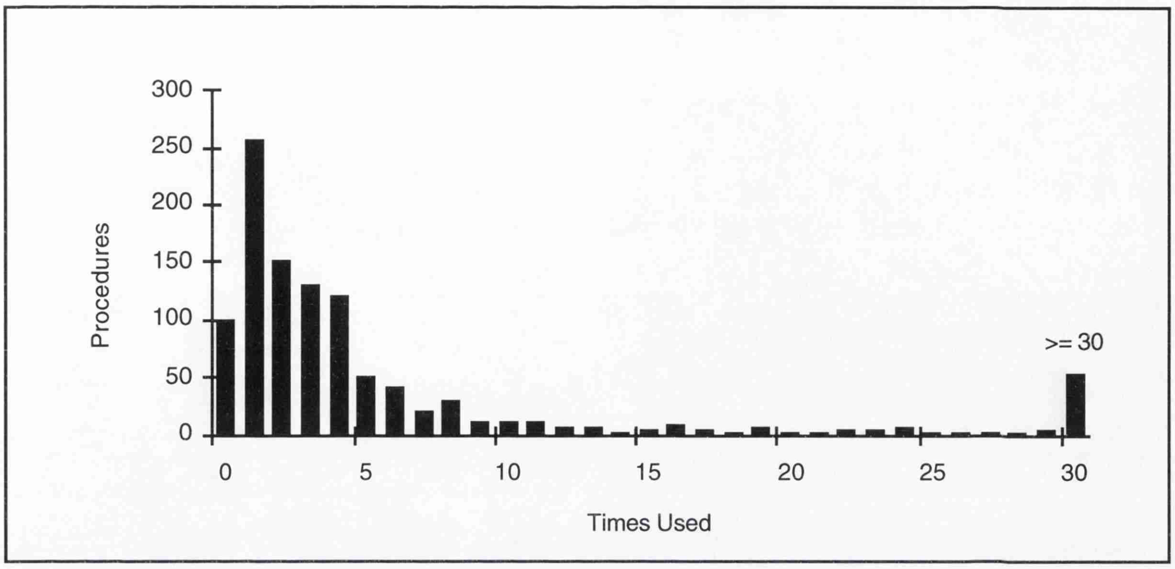


Figure 5.11: Distribution of use of procedures

5.7.8.1 Consequences of Change to Procedures

A procedure can be changed by changing its name, type or value. A change to the value (body) will normally not require any propagation to other parts of the application [Dearle 1987, Connor 1991, Dearle *et al.* 1992, Atkinson 1993, Cutts 1993a] (see Chapter 6). So, in the following text, a change is either a change to the name or to the type of a procedure. Changing a procedure implies that all places of use must be changed accordingly. Table 5.18 shows the use of procedures in the applications. Only procedures that are both defined and used are included. (Changing an unused procedure does not have any consequence, of course.) On average between 3.4 (PartsDB) and 9.2 (Comp/TSIT) places have to be edited. The average use seems to be relatively independent of the number of procedures in the application. The use frequency varies from one (*Min*) to between 11 and 204 places (*Max*).

The argument above should be modified slightly. If a procedure p_1 is changed (excluding renaming), it is not necessary to perform any edit if the occurrences of p_1 are on the right hand side (in right context) of assignments on the form:

let $p_2 := p_1$ (or **let** $p_2 = p_1$)

However, such assignments occur only occasionally in the analysed applications. If there are such cases, however, then the changes must also be propagated to all places where p_2 is used. If p_2, in turn, is also used in right contexts of assignments, then there is yet another level of change propagation, and so on.

Moreover, if the type of a procedure p_3 is changed, and p_3 is passed as a parameter to another procedure p_4, then the call places do not require change. The declaration of p_4 must be changed, however.

Application	Procs	Min	Max	Mean	Std	Sum
Benchmark	8	1	12	4.0	4.0	32
Bibliography	215	1	77	4.9	9.2	1060
Comp/TSIT	145	1	156	9.2	23.4	1341
EcoSys	101	1	51	3.9	7.8	389
ImplADT	20	1	13	4.2	3.9	84
Map	62	1	44	7.4	11.2	458
PartsDB	14	1	11	3.4	3.5	48
WIN	423	1	204	7.1	13.4	2998
Total	988	1	204	6.5	13.8	6410

Table 5.18: Use of procedures

5.7.8.2 Context of Procedures

Table 5.19 describes the context in which the procedures are used. Regarding declarations, 923 are inserted into some environment (in practice that means made persistent), whereas 1538 are declared local to a program (*ValueDecl* and *RecursiveValueDecl*) and are thus made temporary. The *UseClause* row (3385) shows how many times a persistent procedure is brought into the scope of a program. The context values are dominated by *ArgUnaryOpValue* which is mostly procedure calls, but also includes procedure identifiers, not representing calls, occurring on the right hand side of '=' or ':=' in assignments. *PrimFunctionCall* denotes calls to the only built-in procedure *PS*.

Context	Frequency	Percentage
ArgUnaryOpValue	8453	48.4
UseClause	3385	19.4
ValueDecl	1476	8.5
StructFieldDeref	1360	7.8
BindingInserted	923	5.3
StructFieldDecl	551	3.2
Assignment	418	2.4
PrimFunctionCall	358	2.1
ProcParamDecl	206	1.2
ContainsCheck	157	0.9
ADTFieldDeref	100	0.6
RecursiveValueDecl	62	0.4
Total	17449	100.0

Table 5.19: Context of procedures

5.7.8.3 Polymorphic and Specialised Procedures

The previous two sections do not distinguish between monomorphic and polymorphic procedures. This section, however, describes the use of polymorphic procedures and to what extent they are specialised. Determining the consequences of changing a polymorphic procedure also involves measuring the use of specialised procedures. The notion of specialised procedure is best illustrated by an example. Assume that a polymorphic procedure *p* is defined as follows:

```
let p :=  proc[ t ]( x : t )
          begin ...   end
```

The procedure is specialised by instantiating it with a type, e.g. *integer* as in:

```
let pInt := p[ int ]
```

The specialised procedure *pInt* is now an ordinary monomorphic procedure with one parameter of type integer.

In total, 19.3% of all name occurrences are monomorphic procedures and 2.1% are polymorphic (Table B.1). There are, however, significant variations between the applications. For example, Map has 10.7% monomorphic and 10.2% polymorphic compared with 20.3% monomorphic and 0.2% polymorphic in WIN. Even though nearly all polymorphic procedures are made persistent, their use is about 30% lower than the use of monomorphic procedures. One reason for this relatively low use is that most of the polymorphic procedures in the study are provided by the Map application which does not use most of them itself. The intention is that the map constructs should be utilised by other applications. This has only been done to a lesser extent, however, because the Map implementation has by the time of the study (August 1991) just been released. Another reason is that polymorphic procedures are used indirectly in specialisations.

Measurements pertaining to polymorphic and specialised procedures are presented below. Since some of these measurements cannot be obtained from TSIT alone but must be collected by investigating the source code manually, only three applications have been measured: Benchmark, Comp/TSIT and Map. Table 5.20 shows how much the polymorphic procedures are used and how many different types that are used in the procedure calls. A type in this context means a tuple of actual type parameters: [string], [int, env], etc. Benchmark has 9 polymorphic procedures which are used 1.1 times on average. The 20 polymorphic procedures of Comp/TSIT have the largest use frequency (6.3). In Benchmark the same type is always used, whereas the Map procedures are instantiated with up to 17 different types. Comp/TSIT has the largest average (2.5) of different types used per procedure.

		Times Used			Different Types		
Application	**Poly Procs**	**Min**	**Max**	**Mean**	**Min**	**Max**	**Mean**
Benchmark	9	1	2	1.1	1	1	1.0
Comp/TSIT	20	0	37	6.3	1	10	2.5
Map	112	0	16	4.6	1	17	1.8

Table 5.20: Use frequency and number of types instantiated

Table 5.21 shows the extent of polymorphic procedure specialisation. The *PolyProcs* column contains the number of polymorphic procedures that are specialised. The *Spec Procs* column shows the number of specialised procedures – ranging from 7 (Benchmark) to 83 (Map). A comparison with Table 5.20 reveals that respectively 78%, 10% and 16% of the polymorphic procedures of the three applications are involved in specialisations.

Below the *Specialised Procedures* heading, *Min, Max* and *Mean* denote respectively the smallest, greatest and average number of specialisations for one polymorphic procedure. The *Different Types* column describes the number of different types (a tuple as described above) that are used in specialisations of one particular polymorphic procedure. For example, in Comp/TSIT there are on average 6.5 different types that are used per polymorphic procedure.

	Poly	Spec	Specialised Procedures			Different Types		
Application	**Procs**	**Procs**	**Min**	**Max**	**Mean**	**Min**	**Max**	**Mean**
Benchmark	7	7	1	1	1.0	1	1	1.0
Comp/TSIT	2	13	3	10	6.5	3	10	6.5
Map	18	83	1	32	4.6	1	5	2.2

Table 5.21: Specialised procedures

Table 5.20 indicates that a change to a polymorphic procedure would affect between 1.1 and 6.3 places on average (*Times Used, Mean*). In addition, for polymorphic procedures being specialised the changes must also be propagated to the places where the specialised procedures are used, that is, 3.3, 6.8 and 1.0 places in Benchmark, Comp/TSIT and Map, respectively (Table 5.22).[1] In summary, changing a polymorphic procedure used in specialisations will on average affect 4.4, 50.5 and 9.2 places in the respective applications.[2]

[1] In Table 5.22 the context values *BindingInserted* and *ValueDecl* constitute the number of specialised procedures, while a use means an occurrence in one of the contexts *UseClause* or *ArgUnaryOpValue*. So, the use is calculated by dividing the sum of *UseClause* and *ArgUnaryOpValue* by the sum of *BindingInserted* and *ValueDecl*.

[2] The calculation follows:
average use of polymorphic procedures [1.1, 6.3, 4.6]

Application	Usage	Context	Frequency	Percentage
Benchmark	ValueDeclaration	BindingInserted	7	23.3
		UseClause	9	30.0
	RightContext	ArgUnaryOpValue	14	46.7
Benchmark Total			30	100.0
Comp/TSIT	ValueDeclaration	BindingInserted	12	11.8
		UseClause	34	33.3
		ValueDecl	1	1.0
	RightContext	ArgUnaryOpValue	55	53.9
Comp/TSIT Total			102	100.0
Map	ValueDeclaration	ValueDecl	83	49.1
	RightContext	ArgUnaryOpValue	86	50.9
Map Total			169	100.0

Table 5.22: Usage and context of specialised procedures

5.7.9 Measurements Related to Environments

A substantial part of Napier88 code is concerned with operations on environments. Table 5.23 shows the number of identifier occurrences in such contexts. The occurrences of the involved environments themselves are also included (fifth row). It appears that in total 20% of all name occurrences pertain to environments. This proportion may be compared with corresponding measurements in other programming environments. One example is the classical figure in the persistent literature that typically 30% of all code in conventional languages is concerned with transferring data to and from secondary storage [IBM 1978]. Compared with that figure, using Napier88 seems to reduce the volume of code related to secondary storage by about one third.[1]

+ (average number of specialised procedures [1.0, 6.5, 4.6]
* average use of specialised procedures [3.3, 6.8, 1.0])
= total number of affected places [4.4, 50.5, 9.2]

[1] Being precise, these operations apply to environments in general which do not necessarily have to be persistent. In practice, however, there are only a very small number of environments that are only transient (less than 5% in the actual sample of applications).

Context and Environments	Bench-mark	Biblio-graphy	Comp/TSIT	EcoSys	Impl-ADT	Map	Parts-DB	WIN	Total
BindingInserted	70	31	235	31	42	188	15	972	1584
BindingDropped	0	30	11	31	17	7	0	375	471
UseClause	380	999	1614	463	94	644	65	3359	7618
ContainsCheck	0	30	14	31	0	6	0	758	839
Environments	239	431	602	364	102	409	44	4197	6388
Sum	689	1521	2476	920	255	1254	124	9661	16900
% of Total	20.1	14.0	16.9	21.8	9.6	13.2	7.2	25.7	20.0

Table 5.23: Number of name occurrences related to operations on environments

There are 3706 name occurrences of environments associated with the 7618 occurrences in the use-clauses. In total, these 11324 occurrences constitute 13% of all name occurrences which confirms the need for tools that (partly) automate the process of specifying use-clauses (Section 8.2.4).

5.7.9.1 Changes to Environments

A problem experienced by Napier88 programmers is the management of bindings in the persistent store. A change to a program that inserts bindings may have unexpected consequences for other programs that utilise these bindings. The removal of bindings may also have serious impact on other parts of the application.

Application	Programs	Programs with Insert	Bindings Inserted	Per Program	Programs with Drop	Bindings Dropped	Per Program
Benchmark	29	11	70	6.4	0	0	
Bibliography	38	26	31	1.2	25	30	1.2
Comp/TSIT	80	11	235	21.4	9	11	1.2
EcoSys	24	20	31	1.6	20	31	1.6
ImplADT	11	8	42	5.3	3	17	5.7
Map	25	11	188	17.1	3	7	2.3
PartsDB	4	2	15	7.5	0	0	
WIN	156	148	972	6.6	147	375	2.6
Total	367	237	1584	6.7	207	471	2.3

Table 5.24: Programs modifying environments

Table 5.24 shows the number of programs inserting or dropping bindings. The *Programs* column contains the total number of programs in the respective applications. The organisation of the applications with respect to the way environments are being operated on varies significantly. For example, in Comp/TSIT only 14% of programs insert bindings which is in contrast to the 83% and 95% of EcoSys and WIN. The number of bindings inserted per program (average of the programs that actually contain insertions) also differs considerably – with Bibliography (1.2) and Comp/TSIT (21.4) as the two extremes.

The problem of installing and modifying a Napier88 application would be simplified if no program updated more than one environment. How does current practice compare with such a convention? Table B.4 in Appendix B shows the minimum, maximum and mean number of different environments modified (*BindingInserted* or *BindingDropped*) per program. (The same sort of statistics are also provided for *UseClause* and *ContainsCheck.*) The table reveals that in Bibliography and EcoSys, a program never inserts into more than one environment (*Max* is 1) which is in contrast to, e.g., PartsDB and WIN in which the average is 3.0 and 2.7, respectively.

Table B.5 in Appendix B shows how many programs operate on each environment. It appears that bindings are inserted into an environment in 6.4 programs on average. Bibliography is an extreme case in which the same environment is being inserted into in 26 programs. WIN is another extreme with a maximum of 143 programs and 10.2 programs on average. At the other end of the scale are Map and PartsDB whose environments only 1.5 and 1.2 programs are being inserted into, on average.

Most of the removals of bindings occur in the same programs as the insertions, but not always. Table 5.25 shows the number of environments a given program either inserts into or drops from. Table 5.26 shows the number of programs that inserts into or drops from a given environment.

Application	Programs	Min	Max	Mean	Std	Sum
Benchmark	11	1	3	1.5	0.7	16
Bibliography	26	1	1	1.0	.	26
Comp/TSIT	12	1	4	2.3	0.9	27
EcoSys	20	1	1	1.0	.	20
ImplADT	11	1	3	1.4	0.9	15
Map	12	1	4	1.6	0.9	19
PartsDB	2	1	5	3.0	2.8	6
WIN	148	1	13	2.7	2.3	406
Total	243	1	13	2.2	1.8	535

Table 5.25: Environments modified by a program

Comparing Table 5.25 with Table B.4 reveals that only Comp/TSIT, ImplADT and Map have programs that drop bindings from an environment without also inserting into the same environment. (The *Programs* columns of Table 5.25 have larger values than *Programs* columns of Table B.4 for the context *BindingInserted.*) Comparing Table 5.26 with B.5 reveals that for all environments from which bindings are dropped, there are also bindings being inserted (which is reasonable).

Application	Envs	Min	Max	Mean	Std	Sum
Benchmark	5	1	6	3.2	1.9	16
Bibliography	1	26	26	26.0	.	26
Comp/TSIT	9	1	5	3.0	1.7	27
EcoSys	4	2	11	5.0	1.0	20
ImplADT	6	1	3	2.5	0.9	15
Map	11	1	3	1.7	0.8	19
PartsDB	5	1	2	1.2	0.4	6
WIN	40	1	143	10.2	12.0	406
Total	81	1	143	6.6	8.5	535

Table 5.26: Programs modifying an environment

5.8 Summary

The Thesaurus-based Software Information Tool (TSIT) has been built in order to support program development and maintenance in Napier88. The major component of TSIT is the thesaurus which holds information about names in the source programs and names denoting name-type-value-constancy bindings in a persistent store. The thesaurus contains cross-reference information and detailed information about kinds of identifiers, contexts of identifier occurrences, etc. The need for tools providing such information has often been experienced by persistent programmers, for example when TSIT itself was built. Part of the work was to modify the Napier88 compiler. Modifying such a relatively large piece of software requires a thorough understanding of its structure. In the case of the compiler, it was difficult to discover which environments, procedures and other values were required by which compiler components. Discovering the name of the program that inserted those values was also difficult, but necessary in order to locate the source code. In general, the potential success of software reuse depends heavily on the availability and quality of the information about the existing software. A name-based dependency graph like the thesaurus is a useful aid in that respect.

All the thesaurus information is automatically generated and inserted into the persistent store. TSIT itself is also contained in the store.

The TSIT tool was used in a study of the use of names and identifiers in Napier88 programs. A large number of measurements were collected on the basis of thesaurus data from eight Napier88 applications. Some of the measurements were found interesting and have been presented in this chapter.[1] In order to illustrate how system builders use Napier88 and how they organise their software, the following (amongst others) were measured:

[1] Chapter 6 and Appendix B also present measurements relevant to the issues of the thesis.

- the distribution of base types and the type constructors provided by Napier88;
- the proportion of uses versus declarations of type identifiers (an indication of the consequences of changing type definitions);
- the proportion of declarations, left contexts and right contexts of value identifiers;
- the proportion of constants versus variables; and
- the interaction between programs and the persistent store (frequencies of insertion, use and removal of persistent bindings, the number of environments a program operates on, the number of programs an environment is being operated on, etc.).

The sample of the study is too small and specific to conclude that some properties are application dependent and others are programmer dependent. These are examples, however, of hypotheses that could be tested in a more well-defined experiment based on a larger, more representative sample of Napier88 applications.

It has been demonstrated that the thesaurus information can be the source of useful measurements. As will be described in Chapters 6 and 7, the thesaurus information can also be utilised by methodologies and tools for maintaining large, persistent application systems. TSIT will therefore, in addition to the basic persistent technology presented in Chapter 4, serve as enabling technology for the work described in the following two chapters.

Chapter 6

Models and Methodologies

6.1 Introduction

A system development methodology specifies guidelines directing the performance of the various phases of a system development process including analysis, design, implementation, testing, etc. A programming methodology, however, focuses only on the implementation phase. This chapter describes the design of a proposed programming methodology for construction and maintenance of application systems developed in a persistent programming language. It also introduces a *structured persistent application system model* (SPASM) which specifies an architecture for persistent application systems (PASs). That is, SPASM defines a set of constraints that are believed to improve the maintainability of a PAS and which can be both supported and exploited by change management tools. Hence, SPASM describes criteria of the product (the PAS); the methodology describes criteria of the development and maintenance process of that product. SPASM and the methodology mutually support each other (Figure 6.1). Obtaining a PAS compliant with SPASM is simpler (but is still not guaranteed) if the methodology is followed during construction and maintenance.

Both the proposed SPASM and the methodology are general in that they are independent of the actual real-world applications being implemented.[1] They are, however, couched in terms of the persistent programming language Napier88 even

[1] The constraints of SPASM would typically be included in what are referred to as *general integrity rules* of a database, which constrain the application programs, the types (schema) and the collected data (database) and their combination [Date 1990]. Specific integrity rules express constraints in the real-world application; general integrity rules are independent of a specific application but may depend on a data model being used (e.g. the relational data model).

though most of the principles they encode are applicable to any persistent programming environment.

Adhering to the SPASM and methodology may seem awkward for small programs. Nevertheless, it is an investment that will pay off as the PASs become older and larger.

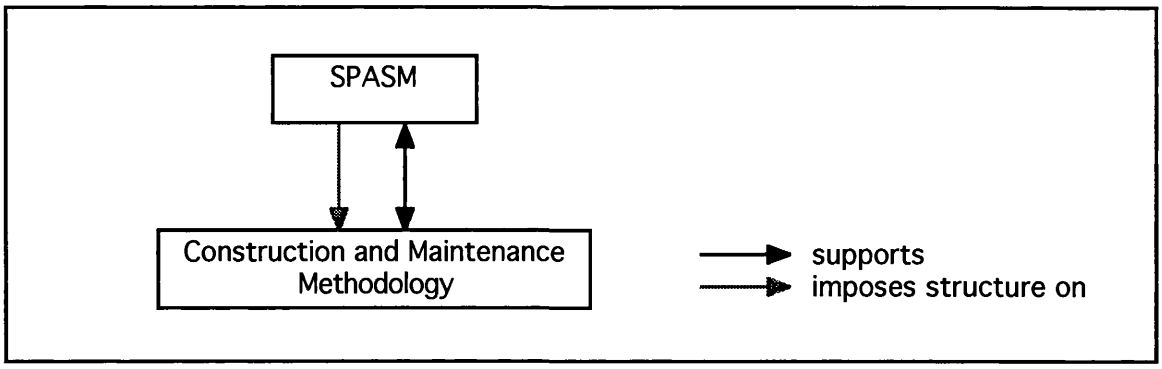


Figure 6.1: Relationship between SPASM and the methodology

6.1.1 Motivation

Traditionally, no specific application models or programming methodologies were proposed for languages like COBOL and FORTRAN, though guidelines or design principles existed. One example is *structured programming* that implies a top-down approach to program development, no use of *goto* statements, etc. [Dahl *et al.* 1972, Jackson 1975]. Another example is the principle of *modularisation* where one should pursue a high degree of cohesion and a low degree of coupling among software components [Constantine and Yourdon 1979].

The interaction among components developed in COBOL and FORTRAN was simple (no program-to-program communication – programs communicated via data files). However, when Ada emerged with its sophistication (multitasking, etc.), the interaction became more complex, and there was a need for tools to manage this complexity, cf. the notion of APSE (Section 3.6.2.1).[1] Napier88 is also a sophisticated language that provides persistence through its ability to operate on environments and was, like Ada, designed with a specific application domain in mind, namely serving as an implementation language for large and long-lived application systems. Experience shows that many untutored novices tend to program in Napier88 in the same style as they previously used in Pascal, C or whatever is their familiar programming language. Even worse, direct misuse of powerful language constructs occurs, which leads to very awkward programs.

[1] Ada was purported to be for embedded systems – "where the computer acts as the controlling device for some larger system" [Sommerville and Morrison 1987].

Programmers who share a common view of how to develop applications in their environment form a particular programming culture. Such cultures may differ considerably from group to group even though the programming language is the same. The rules and conventions of a programming culture *implicitly* express application models and programming methodologies adhered to within that culture. The work described in this chapter *explicitly* formulates such models and methodologies in the context of persistent programming. Some aspects represent rules and conventions in a current Napier88 culture and are already adhered to by experienced Napier88 programmers, e.g. the *persistent location binding methodology*[1] [Dearle 1987, Connor 1991, Dearle *et al.* 1992, Atkinson 1993, Cutts 1993a] in which values (particularly procedures) are updated incrementally without the need for recompilation or re-execution of components referencing these values. Other aspects represent novel ideas also expected to be beneficial for construction and maintenance of PASs.

In addition to supporting development of efficient and consistent PASs, commonly used and explicitly defined models also:

- assist in teaching persistent programming;
- simplify collaboration;
- simplify maintenance of unknown software; and
- permit supporting tools.

Tools above a certain level of sophistication have underlying models which should be made explicit. In compliance with this principle this chapter describes the models that form the basis for the EnvMake tool discussed in Chapter 7.

6.1.2 Requirements for Models and Methodologies

Factors such as programmers, development environments, underlying technology, etc. influence the potential success of the models and methodologies being introduced. Success is more likely to be achieved if the following requirements are obeyed:

i) the programmers should find the models and methodologies intelligible and easy to use;

ii) the models and methodologies should be accompanied by supporting tools;

iii) they should include provision for maintenance; and

iv) there should be a cost/benefit criterion.

1 This is the term used in this thesis, but the methodology is also referred to as an "L-value binding methodology", a "stub methodology", an "incremental construction methodology" and an "application construction architecture".

Developing large and long-lived PASs is a complex and time consuming task. The aims of the models and methodologies are to assist in managing this complexity and increase the efficiency and reliability of the development. It is crucial that the software engineers and programmers find it worthwhile to learn and apply the models and methodologies. It should be easier to fulfil the programmers' tasks by using the models and methodologies; i.e., they should not hinder normal working practice. Nevertheless, programmers must understand that they have to invest in setting up and preserving structure if they want an easier maintenance future.

In order to help achieve the requirement of (i) and to simplify tasks that may be imposed by the models and methodologies (housekeeping operations, checking compliance with constraints and conventions, etc.), appropriate supporting tools must be provided.

Whatever the implementation technology, maintenance is the principal activity for software engineers responsible for a PAS. Maintenance is required to remedy errors, improve existing function and adapt the PAS to its changing circumstances and requirements. The effectiveness of maintenance will be one of the critical factors in assuring longevity. As opposed to traditional system development methodologies [DeMarco 1979, Jackson 1983], the proposed models and methodologies should have an inherent understanding of the nature of evolution in large application systems. Hence, they should provide adequate means for managing change, including necessary consequential change.

6.2 A Structured Persistent Application System Model – SPASM

SPASM is a model of a structure for persistent application systems. It is defined in terms of certain constraints and outlines an architecture for such systems. The consistency of a PAS is evaluated relative to this model; a PAS is partly consistent if it adheres to only some of the SPASM constraints. The reasons why each constraint is included will be explained. These will include the general arguments: simplification of change management, prevention of potential run-time errors, performance improvements, etc. In order to find how these constraints comply with current practice, eight Napier88 applications were analysed (see Section 5.7).

SPASM is concerned with identifying and simplifying the relationships between programs (e.g. source code files) and stored fragments of the PAS (e.g. persistent environments). It focuses on categorising and organising the operations that modify the set of persistent bindings between names and values. The relationship between model, methodology and meta-data will be illustrated; the name information in the thesaurus is used to formulate and verify the SPASM constraints.

The persistent location binding methodology defines a technique for incremental development. Except when type change is involved, programs can be changed independently without the need for recompilation or re-execution of other programs. This methodology is part of the construction and maintenance methodology – the main purpose of the persistent location binding methodology is to support maintenance. SPASM supports this methodology which will therefore be explained before the presentation of the SPASM constraints. The discussion that follows assumes that a PAS compliant with SPASM is being constructed. The reader is reminded that other methodologies would adopt different practices and might still produce viable PASs.

The SPASM constraints are described in detail in Section 6.3. Categories of programs and bindings that are referred to in the definitions of the constraints will be described in the following subsections.

6.2.1 A Persistent Location Binding Methodology

A dummy procedure value, or other types of dummy value denoting an instance of another type, is referred to as a *stub*. A stub for a procedure or another type is initially inserted into a new, persistent location to which other programs can then bind, as an identifier referring to it is declared in a persistent environment. For each stub a template program is created that updates the L-value of the location to hold a useful value.[1] Incremental development is supported in that the template program can be edited and the location correspondingly updated (but its type does not change) with a new L-value without the need for editing, recompilation or re-execution of the other programs using the value.[2]

It is convenient to include a procedure called *uninitialised*[3] as the body of a procedure stub (instead of leaving it completely empty). If a procedure is called before it has been updated, *uninitialised* gives an error message reporting the name of the uninitialised procedure.

Experience has shown that adhering to the persistent location binding methodology is a convenient way of implementing Napier88 applications. To date, almost all stubs in Napier88 applications are procedures. However, as the usage of this methodology

1 In principle, a useful value could be created initially, but it is inconvenient. For example, when changing a procedure body, the "in <env_clause> let" part must be removed from the source program and a use-clause for the procedure must be added. (The original version would still be useful for re-installation or when changing the procedure type.) In contrast, in the interactive hyper-programming environment one can change the procedure body directly by editing the source associated with the procedure (see Section 8.2.6).

2 The stages of this methodology will be explained further in Section 6.5.1.

3 In fact, two dummy values are necessary: *uninitialised_void*: **proc**(**string**) to report premature use of a procedure that returns no value and *uninitialised*: **proc**[*XX*](**string** → *XX*) to report premature application of a procedure that should return a value of type *XX*.

increases, one might expect more widespread use of other kinds too. For example, complex data structures such as symbol tables, tables with geographical information, lists of images, etc. could also be initially created as stubs. The initial values of base type variables are not regarded as stubs. If such variables are made persistent, they may typically be updated in several programs. Hence, the methodology is not feasible for base type values.

6.2.2 Program Categories

A program in this context is a unit of compilation, typically contained in a single Unix file.[1] There are two forms of a program: a source program (identified by the ".N" file naming convention in Napier88, ".c" in C, etc.) and an executable program (identified by the ".out" file naming convention).

In order to define and describe the SPASM model and the methodology to be introduced, it has been found convenient to categorise the programs according to their semantics. On the criteria of how they operate on the persistent store and where they define types, programs are divided into the following categories:[2]

- *Type-program* – a program whose contents are exclusively type definitions.
- *Insert-program* – a program that inserts at least one binding into a (normally) persistent environment but neither updates a persistent location nor drops any binding.
- *Update-program* – a program that updates at least one persistent location but neither inserts nor drops any binding.
- *Drop-program* – a program that drops at least one binding but neither updates a persistent location nor inserts any binding.
- *Startup-program* – a program that uses at least one binding but neither changes the binding to a persistent location, nor inserts or drops any binding.

A type-program contains the source code for the types in a corresponding type database (Section 4.2.1.1).

---

1 In principle, a program may be represented by several files (e.g. assembled first by a pre-processor, held in a source code control system like RCS, etc.) or may be extracted from one file. Nonetheless, the crucial issue in this context is that a file is the unit on which edits are performed when a programmer is carrying out a step in the task of performing a change.

In a hyper-programming environment (Section 8.2.6), an alternative definition would be appropriate since source code (linked to values) is contained in the persistent store.

2 It should be emphasised that when it is mentioned in the text that a binding is inserted, dropped, etc., it is meant in a static sense; that is, the source code contains insert-declarations, drop-clauses, etc. These declarations and clauses could be part of procedures or conditions implying that they are not necessarily being executed when the program is being executed.

An insert-program contains insert-declarations which are of three kinds:

i) declaration of environments;

ii) declaration of stubs;

iii) declaration of other values.

The purpose of an update-program is: first, to assign a useful value to a persistent location that has previously been initialised with a stub; second, to modify that value[1] as the application evolves. In the analysed applications that adhered to the persistent location binding methodology, the number of update-programs is several times greater than the number of other programs.[2]

A startup-program contains at least one use-clause and typically invokes an interactive menu or any persistent procedure. Its distinguishing feature is that it does not change any of the bindings in any persistent environment. A PAS must have at least one startup-program in order to start up an application.

The term *application-program* will be used to denote any program that is not a type-program. Naming conventions for files holding programs of the various categories will be suggested.

6.2.3 Binding Categories

In this context a binding is a name-type-value-constancy quadruple contained in an environment accessible from a persistent root (Section 4.2.3). An environment is a set of such bindings and is identified by its name and the path of environments from the root.[3] Relative to a given PAS there are three categories of bindings:

- *Export binding* – a binding that is defined (i.e., occurs in an insert-declaration) within the PAS with the intention of being used by other PASs. Procedures in libraries such as the Napier88 standard library, WIN and Maps are typical examples.
- *Import binding* – a binding that is used (i.e., occurs in at least one use-clause) but not defined within the PAS. Again, a typical example is procedures of a library.
- *Internal binding* – a binding that is defined within the PAS but is not an *export* binding, i.e., intended for internal use, only.

An *import* binding of one PAS corresponds to an *export* binding of another PAS. An *export* binding may also be used within the defining PAS itself. By definition only *internal* bindings are present in an *internal* environment. The terms *export, import* and

1 One might argue that a value itself cannot be changed; rather, a new value is created in the variable's location. For simplicity, however, common language usage will be adhered to.

2 This distribution will of course change if programmers adhere to a convention of four programs per stub (Section 6.5.1).

3 There are cases where this is not possible (see Section 7.5.2).

internal will be used as a prefix to categorise bindings of various kinds such as *export procedure, internal environment*, etc.

In Figure 6.2 a box represents a collection of bindings of a certain category. The example shows that PAS1 has no *import* bindings but has produced *export* bindings that are used in PAS2 and PAS3. PAS2 has no *export* bindings. PAS3 has *export* bindings but neither PAS1 nor PAS2 uses them.

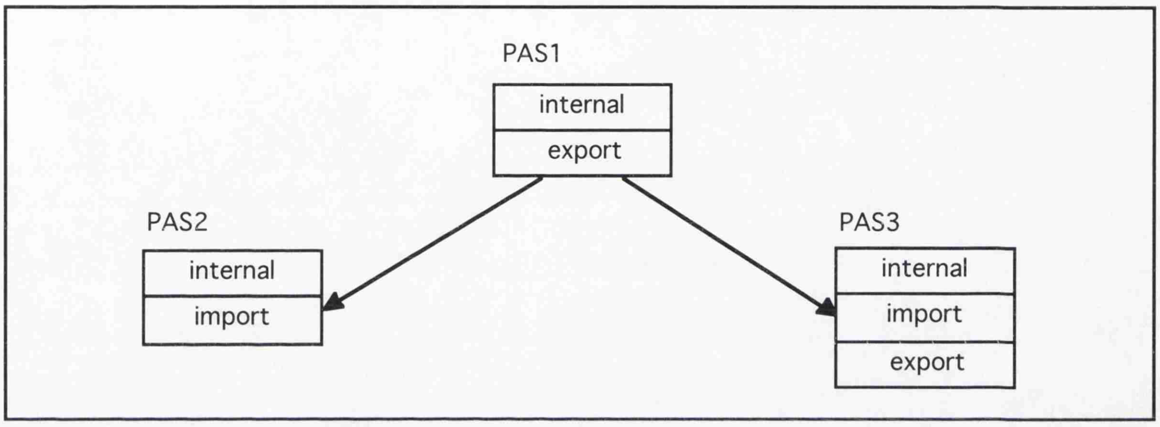


Figure 6.2: Binding categories

6.3 The SPASM Constraints

For a given PAS, the SPASM defines the constraints as shown in Table 6.1. The following sections describe them in detail and provide measurements on how the eight applications referred to in Section 5.7 comply with these constraints.[1]

It is generally difficult (in some cases hardly possible) and invariably time consuming to check the constraints manually. So, a corresponding supporting tool is crucial for the success of the SPASM model. EnvMake is such a tool and is the subject of Chapter 7.

A violation of a SPASM constraint may be an error, or it may just be an indication of a situation that might cause problems – especially in the long run. The software engineering process is improved by adherence to the criteria of SPASM, but sometimes constraint violation may be necessary. Particularly during the initial development, inconsistent states will be normal. Nevertheless, programmers will need reminding of the violation from time to time. Even in the case of violations, the tools should still work correctly and help in developing viable and maintainable PASs (possibly after "consulting" programmers). See further discussion in Section 7.3.2.

[1] These applications were analysed after they were released and used for some time. It is likely that the number of violations would have been greater if the applications had been analysed at various stages during the initial development. On the other hand, if their programmers had been aware of this methodology and had had the benefit of tools proposed in Chapter 7, it is likely that the number of violations at any stage would have been much reduced.

1	**Program categories**
	A program should belong to exactly one of the five categories (Section 6.2.2).
2	**Type definitions**
a)	All type definitions should be used.
b)	All components of a type definition should be used.
c)	A type name should be declared only once within a PAS.
3	**Declaration and use**
a)	An *internal* binding in a use-clause should have exactly one corresponding insert-declaration.
b)	An identifier in an insert-declaration of an *internal* environment should occur in at least one use-clause.
c)	An identifier declared in an application-program (except in insert-declarations) should also be used (either as L-value or R-value) within that program.
d)	If an identifier appears as an L-value in c), then it should also appear as an R-value within that program if the identifier is temporary. If it is persistent and belongs to an *internal* environment, it should also appear as an R-value either within the same program or in another program.
e)	If an identifier is declared as variable, it should appear at least once as an L-value.
4	**Stub constraints**
a)	A procedure inserted as a *variable* should always be a stub.
b)	For each stub declaration there should be exactly one corresponding update-program.
c)	An update-program should update only one persistent location (typically containing a procedure) or a coherent group of persistent locations contained in the same environment.
5	**Drop-clauses**
a)	Only *internal* or *export* bindings should occur in a drop-clause (i.e., *import* bindings should not be dropped).
b)	A binding should occur in maximum of one drop-clause.
c)	A binding in a drop-clause should have exactly one corresponding insert-declaration.
6	**Order of insert- and type-programs**
a)	There should be a partial order among the insert-programs, i.e., no loops among the insert-declarations (Section 6.3.6).
b)	There should be a partial order among the type-programs, i.e., no loops among the type definitions.
7	**Structuring and naming conventions**
a)	There should be a one-to-one correspondence between the structure of directories and environments and between their names.
b)	A program should insert, update or drop bindings of only one environment, and the file containing the program should be stored in the directory corresponding to that environment.
c)	A naming scheme for environments, directories and files with programs should be followed (see Section 6.3.7).
8	**Persistent store**[1]
a)	A binding in a use-clause should be present in the persistent store (unless something else is indicated by the programmer).
b)	A binding in a drop-clause should be present in the persistent store (unless something else is indicated by the programmer).
c)	A binding present in an *internal* environment should occur in at least one use-clause.
d)	A binding present in an *internal* or *export* environment should occur in exactly one insert-declaration.

Table 6.1: The SPASM constraints

[1] The constraints described above involve only source code. The constraints in this group concern relationships between source code and bindings present in the persistent store at the time of analysis.

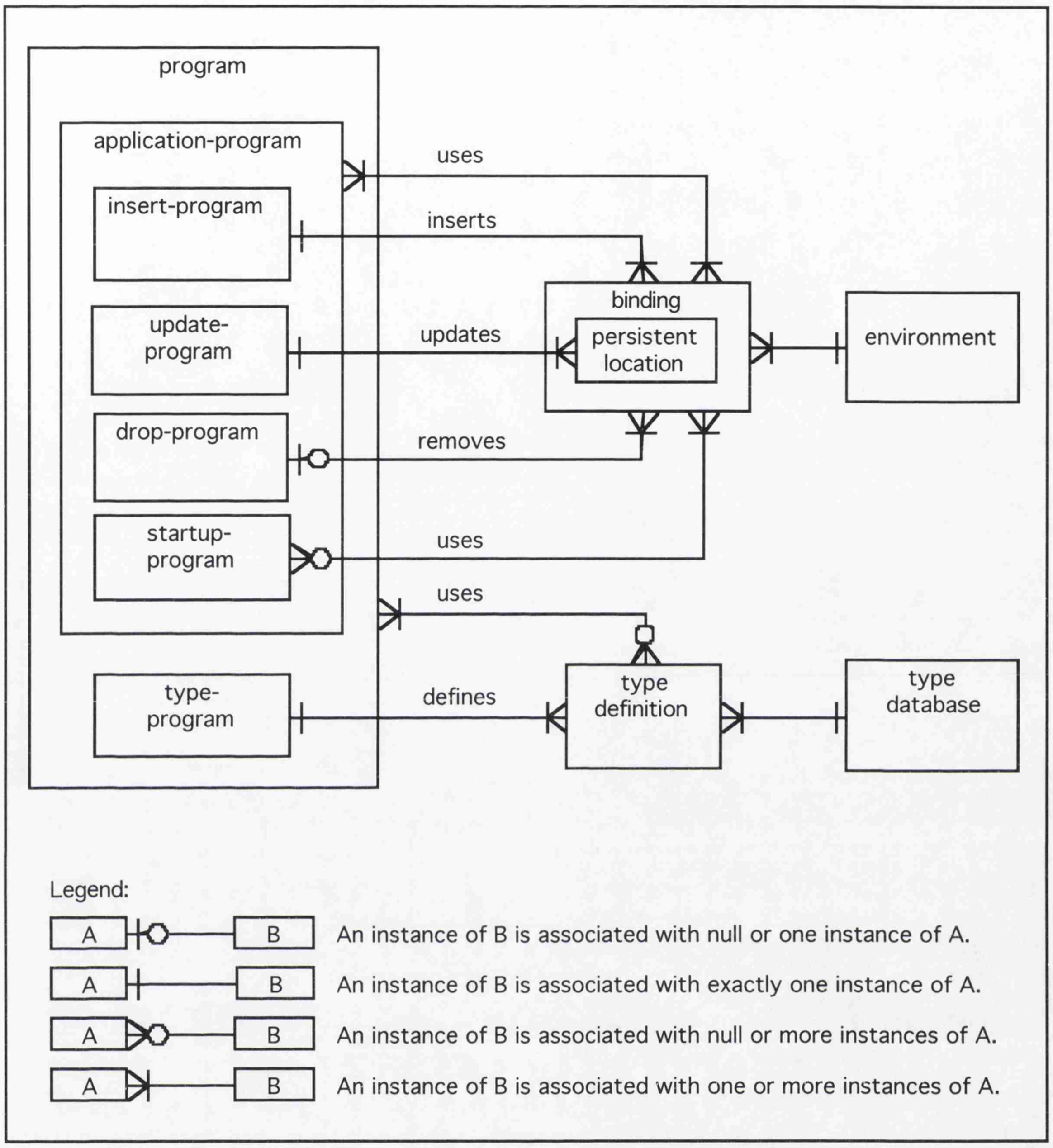

program
application-program
insert-program
update-program
drop-program
startup-program
type-program
uses
inserts
updates
removes
uses
uses
defines
binding
persistent location
environment
type definition
type database
Legend:
A
B
An instance of B is associated with null or one instance of A.
An instance of B is associated with exactly one instance of A.
An instance of B is associated with null or more instances of A.
An instance of B is associated with one or more instances of A.

Figure 6.3: ER diagram of programs, bindings and type definitions

Some of the constraints can be expressed in an Entity-Relationship diagram[1] (Figure 6.3) describing relationships between the type definitions, program categories, persistent locations, environments and other kinds of binding. The *type definition* entity denotes type definitions in type-programs – not those defined locally in application-programs. The arrow texts should be read from the entity on the left of the relationship to the entity on the right. The diagram shows, for example, that a persistent location is associated with exactly one update-program, but one update-program can update several persistent locations.

1 This kind of Entity-Relationship diagram is one of many variants of the original definition [Chen 1976].

6.3.1 Program Categories

Since any program should belong to exactly one of the five categories (constraint 1), only one of the following operations can take place within the same program: declaration of a binding, drop of a binding and update of a persistent location. The main reason for this constraint is to simplify formulation and verification of the other constraints. Another positive effect is that the structure of the actual PAS becomes more intelligible since the programs are categorised according to what they are doing (their semantics).

Measurements

Half of the analysed applications have insert-declarations and drop-clauses in the same program.[1] One of these applications also updates persistent locations in programs that perform insert and drop. Otherwise constraint 1 is complied with.

6.3.2 Type Definitions

Napier88 has structural type equivalence enabling types to be used anonymously, i.e., without any name. Nonetheless, programmers should be encouraged to introduce type definitions, and those required globally should be collected in type-programs where they constitute a useful description of a body of data. In a large PAS there may typically be many type-programs each containing type definitions used in a subsystem.

Unused type definitions and components may confuse maintenance programmers. The application also becomes unnecessarily large and complex which in turn may impair performance and maintainability. Therefore, all type definitions should be used within a PAS (constraint 2a).[2] Also all components of a type definition should be used. That is, the structure fields, variant branches, components of abstract data types, etc. of instances of the type should be de-referenced at least once (constraint 2b).

Constraint 2c states that a type name should be declared only once. Two or more type definitions in different application-programs may violate this constraint in two ways. First, since types may be defined locally in application-programs, then two or more types might be defined with the same name and type (expression) in the overall application. In that case they should be replaced by exactly one definition in a type-program. The same argument applies to equivalent type definitions in different type-programs. Such type definitions should be replaced by one definition in a type-program at a higher level, i.e., the type definitions should be more global.

1 The reader is reminded that these applications were written before SPASM was formulated. The programmers had no supporting tools and had no expectation that their code would be examined.

2 The compiler already checks the inverse – that a type definition is declared either within the program itself or in an associated type database.

Second, type definitions may have the same name but denote different types. To avoid confusion they should then be renamed to acquire unique names.[1]

Multiple declarations of type names are confusing, require unnecessary compilation and are a potential problem with respect to change. Maintaining consistency requires that all declarations describing the same concept (e.g. *Person*) must be changed if the intention is to modify the implementation of the concept (e.g. add a new attribute). It is difficult to arrange that when several programmers (responsible for several components) require use of a common type, they each write out equivalent type definitions (particularly if they are complex). It is even harder to ensure that when the type is amended, the same amendments are applied in every usage context. One concept should therefore be represented by only one type definition.

Measurements

Table 6.2 describes the proportion of unused type definitions in the eight analysed applications. The *Programs* column contains the number of programs in each application. *Type Programs* shows the number of programs that actually include type definitions. There are significant variations among the applications. The principle of ImplADT and PartsDB (and partly Bibliography and EcoSystem) seems to be to declare all types within each program in which they are used, whereas in the other applications, type-programs are used extensively. The last two columns of the table show respectively the total number of type definitions and the number of type definitions that are unused. Only three applications use all type definitions (0 unused types). Benchmark and Comp/TSIT have many unused type definitions (90 of 127 and 100 of 168, respectively). The reason is that when parts of other applications are integrated with the one currently being developed, it is easiest to apply a "maximum approach" with respect to type declarations. That is, all the type declarations of the other applications are copied into the new application. In Comp/TSIT the Napier88 compiler types, TSIT specific types and the Map types are copied. Most of the 100 unused types in Comp/TSIT are part of the Map types; a few of them are compiler specific types not used in Comp/TSIT. This indiscriminate copying of types is probably indicative of a requirement for a tool to collect required items (types or values).

Similar measurements of the proportion of components declared as part of type definitions but never de-referenced within the applications can also be provided by analysing the thesaurus contents. The average use of structure fields and variant tags in the analysed applications is described in Section 5.7.7.1.

[1] The inverse – that several names denote the same type – is accepted. A useful by-product of a tool checking the second group of constraints could be an alias list of such type names.

Application	Programs	Type Programs	Type Decl	Unused Types
Benchmark	29	4	127	90
Bibliography	38	23	122	7
Comp/TSIT	80	3	168	100
EcoSystem	24	13	50	3
ImplADT	11	8	84	0
Map	25	1	116	0
PartsDB	4	4	109	8
WIN	156	5	73	0
Total	367	61	849	208

Table 6.2: Unused type definitions

In applications where types are defined locally in application-programs – rather than being collected in type-programs – the study shows that type definitions tend to be re-defined. Table 6.3 contains the number of type identifiers and type names and the ratio between them. The ratio describes the average number of identifiers per name. Re-definitions do not occur in Map and (practically speaking) not in Benchmark and Comp/TSIT either, whereas in ImplADT and PartsDB the same type name is declared respectively 5.6 and 3.9 times on average.

Measurement	Bench-mark	Biblio-graphy	Comp/ TSIT	Eco-Sys	Impl-ADT	Map	Parts-DB	WIN	Total
Type identifiers	127	122	168	50	84	116	109	73	849
Type names	125	85	162	40	15	116	28	55	626
Identifiers per name	1.0	1.4	1.0	1.3	5.6	1.0	3.9	1.3	1.4

Table 6.3: Relationship between type identifiers and type names

6.3.3 Declaration and Use

Constraint 3a ensures the existence of a corresponding insert-program for each of the *internal* bindings used within the PAS. The constraint is violated if there is not exactly one insert-declaration. If there never was any corresponding insert-program or if it has been removed, re-declaration or declaration at another site as part of a system installation would be impossible.[1] Hence, after a re-installation the run-time error "Cannot find ⟨binding⟩ with type: ⟨type expression⟩" would be given during execution of a use-clause

[1] At present, systems are installed by executing insert- and update-programs (though facilities for copying values directly between stores have been developed [Munro 1993]). However, it should still be possible to re-create a persistent system on the basis of the source code (stores may get corrupted, be remote, be isolated or use different value representations). Furthermore, the source programs serve as documentation for the declaration and usage of the bindings in the store.

specifying ⟨binding⟩. Validation of the constraint plus the automation of build management (see EnvMake in Chapter 7) will prevent this error occurring at PAS run-time.

Several insert-declarations for the same binding may cause confusion and are unnecessary. During an installation, a binding can only be inserted once (under the reasonable assumption that no bindings are dropped during an installation). If several insert-declarations were allowed, there would be a risk of the run-time error indicated by the message "Attempt to re-declare ⟨binding⟩ with type: ⟨type expression⟩".

Any identifier declared in an application-program should also be used within that program. (An exception is identifiers occurring in insert-declarations.) More specifically, the value of a local (i.e., transient) identifier should be accessed within the program (an assignment is not sufficient). However, an assignment of a persistent identifier suffices since its value may be accessed in other programs. Constraints 3b, 3c, 3d and 3e all aim at preventing identifiers from being declared (in the manner indicated by the declaration) if they are not used elsewhere in the application. Even though redundant declarations do not affect the functionality of a program, there are several reasons for why that situation should be avoided:

- An unused identifier might indicate a logical error somewhere. (The intention might have been to use the identifier somewhere but due to a programmer error it is not.)
- Unused identifiers might cause confusion when someone tries to understand the program.
- The programs become unnecessarily verbose.
- For performance reasons – for example, holding unused bindings in the persistent store impairs the performance.

If an identifier is declared without being used, it is not necessarily a mistake. Typically, during initial construction programmers may write the declarations of identifiers before they write the code using those identifiers. In any case, the programmers should be informed about all unused identifiers.

Measurements

All the analysed applications have at least one corresponding insert-declaration for each binding occurring in a use-clause. Duplicated insert-declarations are avoided with one exception. In WIN 27% (180 out of 671) of such declarations were duplicates due to a style of *conditional coding*[1] in order to prevent run-time errors. This implies verbose and

[1] This typically involves writing code that checks whether a binding with a specified name and type is in the store before it is being used. If the store contains a binding with the matching name but not matching type, then drop it and insert a new binding with the correct name and type. A new binding is also inserted if no binding with the matching name is present.

quite clumsy code, but was a way of pursuing safety in the absence of proper methodologies and supporting tools during the initial development of WIN.

Compliance with constraints 3b and 3d has not been measured, but measurements concerning constraint 3c are available. Table 6.4 shows that 7.1 per cent of all declared value identifiers are unused. (Declarations of formal procedure parameters are excluded since a large number of them are declared in dummy procedures – in compliance with the persistent location binding methodology – and are thus deliberately not used within the procedure body.)

Context	Total	Unused	% Unused
UseClause	7618	719	9.4
ValueDecl	6726	302	4.5
RecursiveValueDecl	62	1	1.6
Total	14406	1022	7.1

Table 6.4: Unused value identifiers

Table 6.4 reveals that the majority of unused identifiers are declared in use-clauses. There are several reasons for why this kind of redundancy occurs:

- Large use-clause specifications are copied indiscriminately from other programs.
- Too many identifiers are declared in the belief that they would be needed later.
- Code using identifiers are removed without the programmer remembering to remove the corresponding declarations.

The extent of unused identifiers in use-clauses varies significantly among the applications (ranging from 2.8% to 29%). This range includes the Comp/TSIT application which has an extremely low value (0.5%), but that was due to the use of the EnvMake feature for detecting such anomalies (Section 7.3.1). The measured programs were developed by programmers related to the "Napier88 community". It is reasonable to assume that real-world application programmers without tool support would have an even greater proportion of inconsistent software. In any case, the measurements confirm the need for tools to detect redundant declarations.

More detailed measurements showing the variations between the applications, the kind and context of the unused identifiers, etc. can be found in [Sjøberg 1992].

6.3.4 Stub Constraints

The purpose of the stub constraints is to accommodate the persistent location binding methodology. Emphasis is on procedures even though the methodology can be applied to all kinds of values (Section 6.2.1).

Constraint 4a ensures that a program creating a variable procedure is separated from the program filling it with a useful value (the update-program). Incremental update could have been facilitated even though the insert-program had created the procedure with a useful value initially, but experience has shown that it is convenient to separate the creation and update.[1] If a procedure is not supposed to be updated, then it should be declared as a constant.

Constraint 4b ensures that there actually exists a corresponding update-program for each stub. If no update-program were present, then the value would remain dummy. A dummy procedure, implemented with *uninitialised* (Section 6.2.1), aborts if it is called. Moreover, there should be exactly one update-program since managing several update-programs is error-prone and complicates the application structure unnecessarily.[2]

Constraint 4c aims at enhancing simplicity and clarity by stating that only one persistent location should be updated in a program. Finding this corresponding update-program is easier if the variable and the file have the same name. It may sometimes be convenient to update a group of closely related persistent locations in the same program (in which case the naming convention cannot be followed, of course).[3]

Measurements

Only three of the eight applications insert variable procedures. Table 6.5 shows that a significant number of variable procedures in WIN do not have a corresponding update-program. The reason is that the use of stubs was not commonly adhered to at the time WIN was developed. So, the procedures were assigned "sensible" values when they were initially inserted into the persistent store.

Application	Variable procedures	No update	More than one
Comp/TSIT	84	16	0
Map	122	3	2
WIN	472	76	8

Table 6.5: Update of procedure variables

The 16 procedures not being updated in Comp/TSIT were caused by the lack of knowledge of the compiler application when the author was modifying it to adapt it to the

1 For example, when changing a procedure body, the "in <env_clause> let" part must be removed from the source program and a use-clause for the procedure must be added, but the original version would still be useful for re-installation or when changing the procedure type.

2 As mentioned, the SPASM constraints apply to a given *version* of a PAS. If a programmer wishes several alternative update-programs, then they should belong to different versions of the PAS.

3 Some methodologies would insist on exactly one persistent location being updated within an update-program (Section 6.5.1).

needs of TSIT. Code that updated stubs was removed without removing the corresponding insert-declarations. This may be common when large suites of software are modified (particularly other people's software).

6.3.5 Drop-Clauses

Only *internal* or *export* bindings should occur in a drop-clause (constraint 5a). An imported binding belongs to another PAS (for which it is an *export* binding). Removal should thus only be allowed from within that PAS. At present, standard libraries and other libraries are copied to the programmer's local persistent store. In future, when concurrency and distribution are provided [Munro 1993], the system itself should prevent any attempt at dropping bindings belonging to other PASs.

Constraint 5b is introduced since an application is unnecessarily complex and may cause confusion if there are several drop-clauses for the same binding. For example, if a binding has been dropped, detecting the actual drop-program might be difficult.

Constraint 5c helps prevent the run-time error indicated by the message "Cannot drop ⟨binding⟩ it is not present". If there is no corresponding insert-declaration for the binding, it would not be inserted by any of the programs belonging to the actual PAS.

Measurements

Constraint 5b is violated by the WIN application only and is due to its conditional style of programming mentioned earlier. All the applications comply with constraints 5a and 5c.

6.3.6 Order of Insert-Programs and Type-Programs

The concepts of *partial order* and *topological sorting* will be explained on the basis of [Knuth 1973] which should be consulted for a more detailed description.

In general, a partial order of a set S is a relation between the objects of S which may be denoted by the symbol "$\prec$". The notation $x \prec y$ means that x precedes y. In our context we may have $T1 \prec T2$, where T1 and T2 are type-programs, indicating that T2 depends on T1, i.e., a type definition used in T2 is declared in T1. Another example is related to the use of bindings. If a program P1 inserts a binding used by a program P2, then $P1 \prec P2$. Figure 6.4 shows a diagram of six programs in a partial order. For example, the arrow from P1 to P2 means that $P1 \prec P2$. The programs are in partial order since there are no closed loops in the diagram. If, for example, the program P1 were changed to use a binding inserted by P4, an arrow should be drawn from P4 to P1. In that case P1, P2, P3 and P4 would constitute a loop and thus violate the partial order. The order is partial since there is no ordering between P1 and P6, for example.

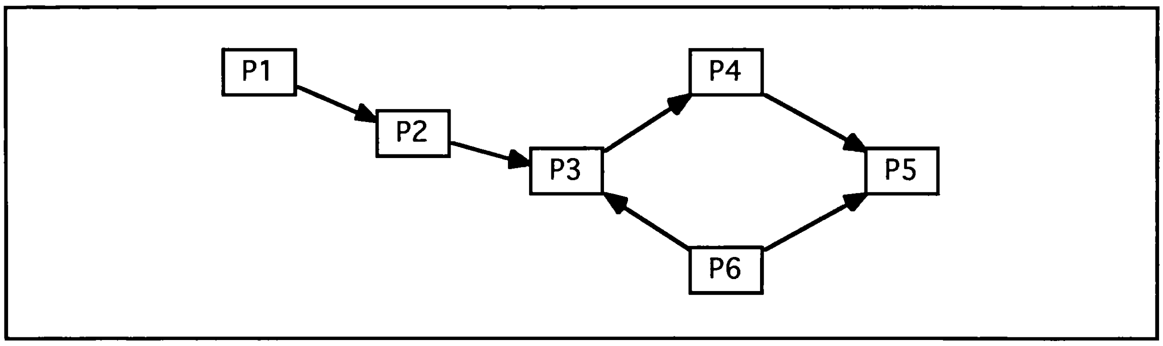


Figure 6.4: A partial order in the set of programs

The process of topological sorting is closely related to partial order [Knuth 1973]:

> The problem of topological sorting is to "embed the partial order in a linear order," i.e., to arrange the objects into a linear sequence $a_1, a_2, ..., a_n$ such that whenever $a_j \prec a_k$, we have $j < k$. Graphically, this means that the boxes are to be arranged into a line so that all arrows go towards the right.

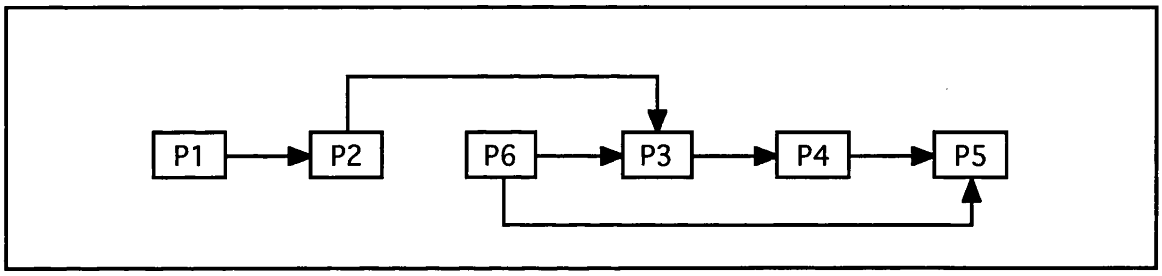


Figure 6.5: Linear sequence after topological sorting

A topological sorting is always possible on a partial order. The result is not necessarily unique – there may be many linear sequences that satisfy the arrangement as described in the quotation. Figure 6.5 illustrates a linear sequence of the programs of Figure 6.4.

System installation requires that the insert-programs are executed in a correct order. That is, the bindings used by one insert-program must already have been inserted by another insert-program before the former can be executed. This is always possible if there exists a partial order among the insert-programs. If the procedures in the insert-declarations contain only dummy bodies (in compliance with the persistent location binding methodology), then all bindings accessed in the insert-programs are environments (except some standard procedures like *date*, which may be used in the creation of an environment, and *uninitialised* and *uninitialised_void*, which may be used in the declaration of a stub).

If neither constraint 1 nor the persistent location binding methodology is adhered to, then the topological sort is particularly useful when installing large systems since bindings may be inserted, updated or used anywhere.

Analogously, the compilation order of interdependent type-programs is significant. A cycle in the use of type definitions could not be processed by the compiler.

There are two issues concerning the partial order:

i) checking that a set of programs has a partial order, and

ii) when possible (see i) discovering any of the linear sequences compliant with the partial order.

As part of checking all the SPASM constraints, EnvMake checks whether the programs of a PAS have a partial order (Section 7.3.1). A linear sequence is suggested (if possible) as an option of an interactive menu of EnvMake (Section 7.2) and as part of EnvMake's features for automatic compilation and installation (Sections 7.4.2 and 7.4.4).

Measurements

The applications were not measured for these constraints, but the author experienced severe difficulties regarding the declaration order during the installation of a modified version of the Napier88 compiler.

6.3.7 Structuring and Naming Conventions

To organise and manage large and complex PASs, certain structuring and naming conventions are necessary. Figure 6.6 sketches the structure of environments in the persistent store for a given PAS. The structure representing current development has a similar structure under the "Error" environment. This isomorphic structure should also be reflected in the file directories since file directories and environments should have the same structure (constraint 7). That is, a root directory should have a corresponding root environment, and for each subdirectory there should be a corresponding subenvironment, and so on. To make this correspondence obvious, a good convention is to use the same name for the respective directories and environments.[1]

The part of a PAS that is in the persistent store and the part that is in the file system are just different representations of the same system. The purpose of the isomorphic structure is to make it easy to discover the correspondences and dependencies between these representations.

[1] Similar structuring and naming conventions could also be introduced for *test* and *release* directories, files and environments.

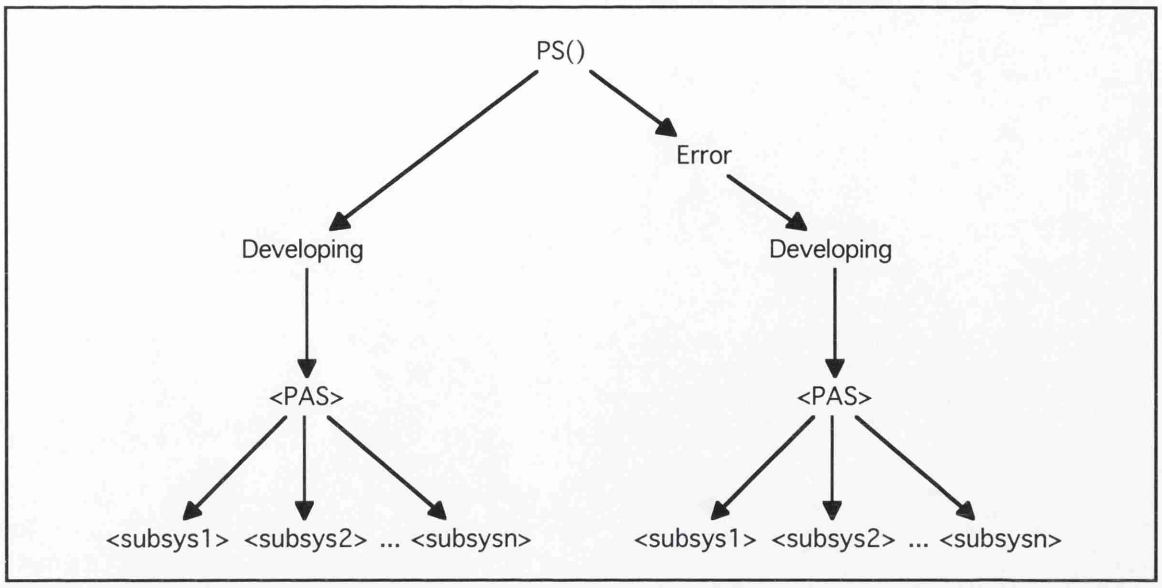

PS()
Error
Developing
Developing
<PAS>
<PAS>
<subsys1> <subsys2> ... <subsysn>
<subsys1> <subsys2> ... <subsysn>

Figure 6.6: Environment structure in persistent store

Files holding Napier88 source programs should have ".N" as suffix. Table 6.6 shows a proposal for naming conventions for programs of the various categories. The naming conventions depend on the methodology to be chosen. If a scheme of four programs per stub is adhered to (Section 6.5.1), the names could respectively be of the form: ⟨binding⟩_insert.N, ⟨binding⟩_update.N, ⟨binding⟩_drop.N and ⟨binding⟩_test.N. Yet another (only slightly different) convention is used in the Napier88 libraries work [Atkinson *et al*. 1993]. The important point is that there *is* a naming scheme – not its exact form.

Program category	Naming convention
type-program	⟨PAS or subsys⟩_types.N
insert-program	i) ⟨PAS or subsys⟩_envInsert.N ii) ⟨PAS or subsys⟩_stubInsert.N iii) ⟨PAS or subsys⟩_dataInsert.N
update-program	Generally "anything"_update.N If only one binding: ⟨binding⟩_update.N
drop-program	⟨envName⟩_drop.N
startup-program	"anything"startup.N

Table 6.6: File naming conventions

Measurements

Two of the applications adhere to the principle of isomorphism (although not 100%) as specified by constraint 7a. One does not comply with it at all, whereas the other applications do it only in part. The ".N" convention is complied with except for three applications that omit the ".N" in their type-programs. The other conventions are mostly new proposals not being adhered to by existing applications.

6.3.8 Persistent Store

The last group of constraints concern dynamic issues in that they involve the actual contents of the persistent store. After system installation all necessary procedure stubs and other persistent values should have been inserted. The look-up of bindings specified in use-clauses are performed at run-time. Hence, failing to find bindings in the persistent store with access path, name, type and constancy as specified in a use-clause will cause a run-time error when the program is executed. Constraint 8a assists in preventing this kind of error. If programs are separately developed, violation of this constraint may be the general case before overall system installation.

The same argument applies to bindings occurring in drop-clauses. Complying with constraint 8b reduces the chances of attempting to drop a binding not present in the store.

During development and *ad hoc* programming, unused bindings tend to accumulate in the persistent store since programmers tend to forget to remove them. Typically, new versions of bindings are inserted (due to, for example, type changes or changes in subsystem structure) without the obsolete versions being removed. Constraint 8c ensures that the persistent store is tidied up in compliance with the current use-clauses in the source code. Collecting such obsolete bindings may be regarded as a form of garbage collection where "garbage" is defined in terms of failure to comply with source code, whereas conventional garbage collectors operate on the persistent store only and define garbage in terms of unreachability from a persistent root. So, a binding in the store that is not referred to in the source code should possibly be removed. However, it could be the case that the source code was changed or a source program deleted by accident. Hence, it is impossible to automate this process entirely without any user intervention, but a warning of the case would undoubtedly be useful.

Bindings in the *export* environments of an application are exempt from constraint 8c since they are not primarily created with the intention of being used within the application itself.

Constraint 8d concerns compliance between the contents of the persistent store and the insert-programs. If a binding, not imported, in the store does not have any corresponding insert-declaration, then the insert-program must have been changed or deleted by mistake, or the programmer must have forgotten to drop the binding when the code was deliberately changed.

6.4 Actions to Conform to the SPASM Constraints

There are an infinite number of kinds of change that may cause violation of the SPASM constraints (though many violations will not occur if certain methodologies are followed). For example, constraint 1 would be violated if an insert-declaration is added to a type-, update-, drop- or startup-program, or if a drop-clause is added to a program that is not a drop-program, etc. There is no point in attempting to describe a plethora of possible causes; it suffices that the programmer knows the kind of violation, where it occurs and the possible actions to rectify the inconsistent states.

For each violation of a SPASM constraint, Table 6.7 describes one or more actions to be undertaken in order to re-establish conformance with the constraint.

No.	Violation	Action to resolve the violation
1	i) An insert-declaration or a drop-clause is contained in an update-program.	Move the insert-declaration or drop-clause to an (existing or new) insert- or drop-program, respectively.
	ii) An insert-declaration and a drop-clause are contained in the same program.	Split them – move (say) the insert-declaration to an (existing or new) insert-program.
2a	There exists a type definition that is never used.	Modify or create a new program that will use the type definition, or delete the type definition.
2b	There exists a component of a type definition that is never used.	Modify or create a new program that will use the component of the type definition, or delete the component.
2c	i) A type declared with same name and type is declared more than once.	If the type is declared in a type-program and the duplicate is in an application-program or in the type-program of a subsystem, then delete the duplicates. If the type is declared in more than one application-program, replace these definitions with one definition in a type-program.
	ii) The same type name is used to declare different types.	Inspect the definitions with the intention of creating unique names.
3a	i) A binding of an internal environment occurring in a use-clause does not have any corresponding insert-declaration.	Delete the use-clause if the binding is not used in the program, or create a corresponding insert-declaration for the binding in an (existing or new) insert-program.
	ii) A binding of an internal environment occurring in a use-clause has more than one corresponding insert-declaration.	Delete all but one of the insert-declarations.
3b	An identifier in an insert-declaration is not used.	Modify or create a new program that will use it or delete the declaration.
3c	An identifier is declared in a program without being used within that program.	Delete the declaration or use the identifier.
3d	i) A temporary identifier occurs only as an L-value.	Remove the declaration of the identifier, or create an R-value occurrence within the same program.
	ii) A persistent *internal* identifier occurs only as an L-value.	Remove the declaration of the identifier, or create an R-value occurrence in the same or another program.
3e	A variable does not appear as an L-value.	Remove the variable or use it as an L-value.

Table 6.7: Actions to reconform to constraints that have been violated (continues)

No.	Violation	Action to resolve the violation
4a	A procedure with "useful" body appears in an insert-declaration.	Create an update-program for the procedure where the "useful" body is being assigned the procedure identifier. Replace the body in the insert-declaration with an appropriate dummy one.
4b	i) There exists no update-program for a stub.	Create a corresponding update-program or delete the stub.
	ii) There exists more than one update-program for a stub.	Keep one of the update-programs.
4c	More than one (major) procedure (possibly in different environments) are updated in the same program.	Create a separate update-program for each of the major procedures with the same name as the procedure (plus the ".N" suffix). Store the program file in the directory corresponding to the environment of the procedure.
5a	There exist more than one drop-clause for the same binding.	Keep one of the drop-clauses.
5b	There exists one or more drop-clauses for an imported binding.	Delete the drop-clauses.
5c	i) A binding of an internal environment occurring in a drop-clause does not have any corresponding insert-declaration.	Delete the drop-clause if the binding is not used in the program, or create a corresponding insert-declaration for the binding in an (existing or new) insert-program.
	ii) A binding of an internal environment occurring in a drop-clause has more than one corresponding insert-declaration.	Delete all but one of the insert-declarations.
6a	There exists a loop among the type definitions.	Inspect the type definitions and resolve the loop.
6b	There exists a loop among the insert-declarations.	Inspect the insert-declarations and resolve the loop.
7a	The directories and environments of the PAS are not isomorphic.	Create programs for appropriate reorganisation and renaming in such a way that no information is lost.
7b	i) A program inserts, updates or drops bindings associated with more than one environment.	Split the program in such a way that the new programs only operate on one environment.
	ii) At least one file containing an insert-, update- or drop-program is not stored in the directory that corresponds to the environment operated on by the program.	Move the file to the appropriate directory.
7c	The naming scheme is not fully complied with.	Rename accordingly.
8a	A binding occurring in a use-clause is not present in the persistent store.	If the programmer is not certain that the binding will be inserted, then add a corresponding insert-declaration to an existing or new insert-program, or delete the use-clause.
8b	A binding occurring in a drop-clause is not present in the persistent store.	If no program will insert the binding, then delete the drop-clause.
8c	An unused binding is present in an *internal* environment.	Add a corresponding use-clause to an existing or new program, or drop the binding from the persistent store.
8d	i) A binding present in an *internal* or *export* environment does not have any corresponding insert-declaration.	Create a corresponding insert-declaration, or drop the binding from the persistent store.
	ii) A binding present in an *internal* or *export* environment has more than one corresponding insert-declaration.	Drop all but one of the insert-declarations.

Table 6.7: Actions to reconform to constraints that have been violated (continued)

6.5 Future Development of a Maintenance Methodology

A persistent maintenance methodology is a model for the process of maintaining PASs. Programmers inevitably need to maintain a PAS by adding new functionality, improving performance and correcting errors. New functionality may require new or modified subsystems and tasks that typically involve adding, removing, renaming or moving files and directories. In a persistent programming environment similar operations would apply to programs and other objects in the persistent store. Required changes to the type definitions or the schema of a PAS may have major impact on other parts of the schema, on the extensional data and on the application programs (Section 2.3). In a higher-order persistent language, where programs are typically represented in the form of procedures in the persistent store, changing the type of a procedure (e.g. adding a parameter) is also a kind of change that potentially requires extensive consequential change.

A maintenance methodology should provide guidelines for how to carry out various kinds of maintenance tasks in a safe and efficient way. Adding a subsystem or changing a collection of type definitions are two examples. The methodology must have an inherent notion of correctness or consistency; its purpose is to guide software builders and maintainers to perform all necessary and no unnecessary changes in a consistent way according to some model of consistency. SPASM (Section 6.3) is an example of such a model in a persistent programming environment.

A problem of designing a programming methodology is to determine its level of detail. For example, experienced programmers may only need a high-level description of the actions to be undertaken for a given kind of change, whereas novices may also wish detailed descriptions. A programming methodology accommodates and makes explicit knowledge and experiences of sophisticated programmers. The support given by the methodology to novices and other programmers at various levels of sophistication may thus compensate for some of their lack of experience.[1]

The problem of details is present, for example, in that a maintenance methodology should help ensure maintainability by preventing deteriorating structure and inconsistencies. In the extreme case, the methodology could specify a list of all conceivable precautions to be taken into account before any change is carried out. For example: "if a file is to be deleted, check what kind of program it holds; if it is a type-program, check that none of its type definitions are used in any program; if it is an insert-program, check, first, that the inserted binding is not used in any other program; second, that the binding is not present in the store; third, etc. etc." It is unlikely, and probably undesirable, that programmers would make all the effort needed to carry out such detailed checks. Provided inconsistent states can be rectified, it is more efficient to perform a

1 A methodology could therefore be outlined at different levels for different categories of programmers.

bulk check of the consistency (the SPASM constraints) at certain stages – preferably automatically by a tool.[1] For changes with potentially serious consequences (e.g. schema changes) one may want to indicate a change and perform the check on the state after the change but before committing the change [Jacobs and Hull 1991, Waller 1991].

The following sections propose an illustrative outline for a maintenance methodology by describing necessary actions to be carried out when changing the type of persistent procedures, when adding or removing subsystems or when changing the schema. The reader is reminded that the outline is a tentative proposal that is as yet unevaluated and unsupported by tools.

6.5.1 Modifying Procedure Types

Procedures, representing programs in persistent store, are commonly created and maintained by Napier88 programmers in compliance with the persistent location binding methodology. As mentioned (Section 6.2.1), any kind of (complex) data value can be the subject of that methodology and be stored in strongly typed persistent locations. Incremental update of values in persistent locations is relatively simple as long as the type and name are unchanged. It suffices to edit, recompile and re-execute the corresponding update-program; editing, recompiling or re-executing any other program is unnecessary. In contrast, renaming or changing the type of an existing binding has greater impact. It involves dropping the old location, creating a new location and changing and recompiling all the programs that use the binding. For example, a procedure's type – its signature – can be modified by a change to the number or types of the formal parameters of the procedure or by a change to its result type.[2]

Three strategies for how to manage changes to the type of an L-value binding are presented in respectively Figures 6.7, 6.8 and 6.9. For a given (sub)system, the *"Organisation"* paragraph describes the programs that insert stubs[3] and the programs that insert other bindings. Possible drop-programs are also described. *"Transaction"* describes the actions necessary to carry out the change.

[1] However, even a detailed precautionary check might be feasible if it is automatically and quickly performed by a tool.

[2] When in common parlance saying that "a procedure changes its type," it effectively means in a strongly typed language that the location containing the procedure is dropped and a new one is created, which may hold procedures of the new type. So, from the system's point of view, one procedure has been deleted and another one has been created. From the programmers point of view, however, it is the same procedure (identified by the same programmer-introduced name) which has changed.

[3] A stub declaration creates a persistent location initialised with an L-value binding to a dummy value.

Organisation

i) All insert-declarations are collected in one file (`insert.N`).

ii) For each stub declaration there is exactly one corresponding update-program. An update-program, in turn, updates only one persistent location or a coherent group of persistent locations contained in the same environment.

Transaction

i) Drop the existing binding by creating, compiling and executing an *ad hoc* drop-program.

ii) Edit the existing insert-declaration in `insert.N` to accommodate the new type.

iii) Create an *ad hoc* insert-program by copying the use-clauses and the newly changed insert-declaration from `insert.N`.

iv) Compile and execute the *ad hoc* program and then delete it.

v) If it is an L-value binding, edit, recompile and re-execute the corresponding update-program.

vi) Edit, recompile and re-execute all programs that used the old binding.

Evaluation

+ Works for any kind of binding – L-value or R-value.

+ The file `insert.N` contains all necessary insert-declarations for the (sub)system – this simplifies installation.

+ Efficient – no unnecessary drop or insertion.

+ Relatively few files in the directory – easy management.

- The emphasis on *ad hoc* programs may lead to errors and inconsistencies (copy wrong code, etc.) and difficulties in reconstructing the actions.

- Unsuitable for automation.

Figure 6.7: Strategy 1

All three strategies can be applied in any persistent language providing first-class procedures that can be stored in strongly typed persistent locations. At present, Napier88 programmers adhere to either strategy 1 or strategy 2. Except for the update-programs corresponding to the L-value bindings, there is no difference between L-value and R-value bindings in strategy 1. Although efficient, the use of *ad hoc* programs in strategy 1 is error-prone and results in poor documentation; not even source code documents the actions that have been carried out. Strategy 2 represents a more organised alternative but is extremely inefficient.

Organisation

i) All insert-declarations of stubs in one file (`stubInsert.N`); all insert-declarations of other bindings in another (`dataInsert.N`).

ii) For each stub declaration there is exactly one corresponding update-program (as strategy 1).

iii) There is a drop-program (`stubDrop.N`) corresponding to all L-value-bound persistent locations.

Transaction

i) Drop the existing L-value bindings by executing `stubDrop.N`.

ii) Edit the existing insert-declaration in `stubInsert.N` to accommodate the new type.

iii) Insert the new and re-insert the old bindings by compiling and executing `stubInsert.N`.[1]

v) Edit and recompile the program that updates the changed binding. Re-execute that and all the other update-programs of the other bindings.

iv) Edit and recompile all programs using the changed binding. Re-execute all programs using any L-value binding.

vii) Modify (if necessary) the drop statement for the changed binding in `stubDrop.N`.

Evaluation

+ Having only two insert-programs (`stubInsert.N` and `dataInsert.N`) simplifies installation and bulk insertion.

+ Relatively few files in the directory makes management easy.

- Works only for L-value bindings with corresponding update-programs.

- Inefficient – many unnecessary drops, insertions and updates, i.e., unnecessary compilation and execution.

- Unsuitable for automation.

Figure 6.8: Strategy 2

Strategy 3 is a new proposal whose potential success depends on corresponding tool support (a proposal is outlined in Section 7.6).[2] To illustrate, in the analysed Napier88 applications (Chapter 5) 1015 different bindings[3] were inserted in total. Four files per binding would give a total of 4060 files. Leaving out test programs (since such programs were not included in the analysis either) there would have been 381 files per application on average – compared with 46 in the analysed applications. Tools are therefore clearly needed for program management, automatic program generation and build management (Section 7.4). Moreover, it would be awkward for the programmer to repeatedly edit the

[1] For simplicity of the text, executing a ⟨name⟩.N program should, of course, be read as executing the corresponding ⟨name⟩.out program.

[2] The current Napier88 libraries [Atkinson *et al.* 1993] are being implemented according to this strategy.

[3] Bindings are counted as unique (environment name, binding name) combinations.

needed use-clauses for each binding. Such specifications may be generated by future tools (Section 8.2.4) or could be replaced by "hyper-references" (Section 8.2.6).

Organisation

Four files per binding: ⟨binding⟩`_insert.N`, ⟨binding⟩`_update.N`,[1] ⟨binding⟩`_drop.N` and ⟨binding⟩`_test.N`.

Transaction

i) Drop the existing binding by executing ⟨binding⟩`_drop.N`.
ii) Edit the insert-declaration in ⟨binding⟩`_insert.N`.
iii) Insert the new binding by compiling and executing ⟨binding⟩`_insert.N`.
v) If it is an L-value binding, edit, recompile and re-execute ⟨binding⟩`_update.N`.
iv) Edit, recompile and re-execute all programs using the changed binding.
vii) Modify and recompile ⟨binding⟩`_drop.N` (if necessary).

Evaluation

+ Works for any kind of binding – L-value or R-value.
+ Efficient – no unnecessary drop or insertion.
+ Suitable for automation.
- A vast amount of files to be edited and managed; even with appropriate naming conventions, tool support is crucial.

Figure 6.9: Strategy 3

6.5.2 Modifying Directories and Environments

A (sub)system of a PAS written in a language like Napier88 is represented by a directory in the file system and a corresponding environment in the persistent store. In addition, there may be a corresponding error directory and a corresponding error environment. For each of the modifications accomplished for a directory, similar changes should be performed on corresponding environments (and vice versa). Any change (addition, modification or removal) to the tasks of a subsystem will typically be reflected in the files of the directories and in the contents of the environment corresponding to the subsystem.

For example, if a subsystem has ceased to be used, the corresponding directory (including its files) should also be removed since unused components make the PAS unnecessarily large and complex.[2] If the directory was not removed, it would require maintenance like editing the code that uses global[3] type definitions that are subsequently

[1] There is no ⟨binding⟩`_update.N` program if the binding is an R-value.

[2] It is assumed that convenient back-up routines are in effect.

[3] Global type definitions refer here to those that are defined in the overall PAS or in a subsystem at a higher level in a possible hierarchy of subsystems.

changed. Leaving inconsistent code around should be avoided even though it is (at least temporarily) unused. However, before removal a check should be made to verify that none of the bindings in the corresponding environment are referred to in other subsystems. If this is the case, either leave the system as it is, or move the insert- and update-programs operating on these bindings to another subsystem.

If the construction methodology has been strictly followed, no type definitions should be used in other subsystems.

The environment is dropped by creating and executing a corresponding drop-program. If there are other references to that environment, however, it will not be removed from the persistent store. Similarly, values (including L-values) in that environment will be retained if they are referenced. That is, the environment will not be accessible via a use-clause of a program any more, but it will be accessible from the objects already referencing the environment (i.e., no dangling references will occur). So, if the programmer wishes to remove references to the environment as well, this must be done explicitly by tailored programs. The thesaurus information (Chapter 5) could give advice on possible references (e.g. identifying all of the programs using that environment).

6.5.3 Modifying Types – Schema Evolution

In a language with structural type equivalence, programmer-introduced type identifiers are just tokens or abbreviations for types introduced for the convenience of the programmers. In languages where types can be declared in any program (e.g. Napier88), there is no concept of a *schema*, like *database schema* in database systems, which necessarily contains all the type definitions used in a (sub)system. Nevertheless, programmers should be encouraged to collect all the type definitions of a PAS or subsystem in one file – a type-program (Section 6.2.2). The set of type definitions in such a file and the corresponding set of types in a type database are two representations of what will be referred to as a schema.

As stated earlier in Section 3.2, the consequences of schema changes are divided into three categories:

i) Effects on other parts of the schema

ii) Effects on instances – extensional data

iii) Effects on application programs

The consequences of adding, renaming and deleting a type definition are described in Table 6.8. Renaming may not be regarded as a basic schema change; it can be viewed as decomposition of a deletion followed by an addition. The *activities* to be carried out, however, are different and much simpler than the combination of the activities for deletion followed by those for addition. The effect may not be so far reaching since no structure is modified, but it will still affect all the places where the renamed type is being used. Renaming will therefore be regarded as a schema change in this context.

Change	**Action on schema**	**Action on extensional data**	**Action on application code**
Add type definition	Add the new type definition to the type-program in the actual (sub)system.	No action is necessary.	At least one program should declare instances of the new type and at least one program should use them.[1]
Rename type definition	Replace all occurrences of the old type name with the new name in the type-program.	Due to structural type equivalence existing instances in the persistent store can remain unchanged.	Replace all occurrences of the old type name with the new name in the application-programs.
Delete type definition	Delete the type definition from the type-program. If the type name is used in another type definition, then that type definition, in turn, must either be deleted or changed, and so on.	In principle, delete existing instances of this type from the persistent store. However, a warning should be generated (if possible) in the case instances exist.	Delete all declarations in the code where the type definition occurs and all places where instances of this type occur.

Table 6.8: Impact of adding, renaming or deleting a type definition

There are several strategies for tackling the problem of inconsistency between instances of the old type and the new type definition. Some ideas of approaches in the context of a strongly typed language have been presented in [Atkinson 1993] (see also Section 3.2). One should note, however, that in a language with structural type equivalence, the problem of consistency between type definitions and instances is purely at the conceptual level – from the modelling point of view. From the language point of view, instances are bound to *types*, not to *type definitions*. However, that does not make the problem less important. This is discussed further in [Connor *et al.* 1990].

The intention of this section was not to describe detailed strategies for how to manage schema or type changes but to present an overview of issues that must be dealt with in further research in this area (Section 8.2.1). Table 6.8 outlines what kinds of change and change consequences that must be included in an analysis. A similar table should be created for changes to a type definition. For each kind of type change (adding a field, changing the type of a field, delete a field, renaming a variant field, adding a variant branch, etc.) the consequences for the schema, extensional data and application code should be analysed.

[1] This is presumably the motivation for the type change in the first place.

6.6 Summary

This chapter has introduced two models applicable to persistent applications systems and to their construction and maintenance:

- SPASM – a model for the PAS to be built.
- A construction and maintenance methodology – a model for how the PAS should be constructed and maintained.

SPASM is a model for organising persistent application systems. It defines a set of constraints concerning definitions and uses of types, operations on persistent objects (including programs), redundant declarations, unused values, etc. Consistency is defined relative to this model; i.e., an entirely consistent application is one that complies with all the constraints of SPASM.

The SPASM constraints are based on the knowledge of experienced Napier88 programmers and should thus be useful to most programmers in most situations. Novices would particularly benefit from rules for how to organise their persistent applications. However, some constraints might be undesirable in certain circumstances, and there might be a need for additional constraints in other circumstances. Some flexibility should therefore be allowed for when designing tools that support the model. (An analysis of eight Napier88 applications shows that they comply with most of the constraints.) There is a trade-off between flexibility and discipline. Software builders may feel that SPASM constrains their personal programming styles. However, in order to develop complex and long-lived PASs, with possibly many people involved, it is crucial that commonly agreed practices and conventions are used. Standardisation may eliminate peculiar programming styles and may simplify collaboration, maintenance, software reuse, etc. SPASM is an initial suggestion that needs evaluation through use.

The construction methodology guides the software builders in constructing applications systems in compliance with SPASM.

As part of the maintenance methodology, strategies for accommodating a method of programming based on programs bound as L-values to persistent locations have been outlined. A new strategy was proposed whose potential success relies on appropriate tool support. The maintenance methodology also includes a classification of type changes corresponding to schema evolution in database systems. For each kind of change, the methodology presents a high-level description of the necessary steps to be undertaken in order to accommodate the change in a consistent way.

One might argue that understanding and adhering to the SPASM model and the construction and maintenance methodology are a heavy burden to impose on software engineers and programmers. However, developing and maintaining large and long-lived PASs are complex tasks, and one has to invest in adapting to suitable working practices in order to accomplish such tasks. The possibly extra effort required in the short-term should pay off in the long run.

A significant improvement in the programming process will be achieved if adherence to models like SPASM is checked automatically by a supporting tool. Similarly, accompanying tools should actively support, i.e., *partly* automate, construction and maintenance activities as defined by the methodologies. Such tools are the subject of the next chapter.

Chapter 7

EnvMake – A Persistent Programming Tool

7.1 Introduction

The success of the methodology and the SPASM constraints discussed in the previous chapter depends heavily on supporting tools. This chapter describes EnvMake which is a proposal for such a tool. It supports application construction and maintenance in compliance with the persistent location binding methodology (Section 6.2.1). In particular, it checks whether a PAS adheres to the SPASM constraints.

The methodology described in the previous chapter supports incremental development. Programs in the persistent store, represented as procedures, can be changed (provided their types do not change), recompiled and re-executed without the need for any operations on the dependent programs. If the procedure type is to be changed, then the dependent programs must be edited, recompiled and re-executed as well. Similarly, if a program with only type definitions changes, then all dependent programs must be at least recompiled, usually also edited. Automatic assistance in determining and initiating the necessary recompilations and re-executions is crucial. Traditionally, Unix programmers use Make as the supporting tool. All dependencies have to be inferred manually, however, and any change in the dependency structure requires a programmer to edit the Makefile correspondingly. EnvMake is a language specific alternative, written in and for Napier88, that automatically infers the necessary dependencies from the thesaurus information. Hence, there is no need for any file like the Unix Makefile.

Dependency tables used by EnvMake in the check of the SPASM constraints and by the build management features can also be browsed directly by a programmer. The information obtained is useful for the understanding of the application's structure – particularly for large and complex applications.

EnvMake can be run in several modes determined by a parameter after the command as summarised in the following table.

Command	**Function**	**Section**
envMake	Obtain menu giving structural information	7.2
envMake consistency	Obtain warnings of SPASM violations	7.3
envMake plan	Show compilation and execution plan	7.4.1
envMake compile	Perform all necessary recompilations	7.4.2
envMake run	Perform all necessary recompilations and re-executions	7.4.3
envMake install	Perform all compilations and executions necessary for installation	7.4.4

Table 7.1: Parameters of the envMake command

7.2 Information about Application Structure

Programmers can already obtain information about dependencies between names used in an application system through one of the query interfaces of TSIT (Chapter 5). In the case of dependencies between environment operations and the corresponding bindings, EnvMake provides two alternative ways of presenting the information: as *dependency tables* and as *matrices.*

```
INSERT-PROGRAM                    BINDING                    UPDATE-PROGRAM
---------------------------------------------------------------------------
...                               ...                        ...
Library/KeyChooseLib_stub.N       Library\KeyChooseLib       Library/KeyChooseLib.N
Library/MakeDummyLib_stub.N       Library\MakeDummyLib       Library/MakeDummyLib.N
Library/WriteLibName_stub.N       Library\WriteLibName       Library/WriteLibName.N
Person/FindPersonDepend_stub.N    Person\FindPersonDepend    *
Person/MakeDummyPerson_stub.N     Person\MakeDummyPerson     Person/PersonValues2.N
Person/WritePersonName_stub.N     Person\WritePersonName     Person/PersonValues1.N
...                               ...                        ...
```

Table 7.2: Insert-update dependency table

Table 7.2 shows some insert-programs, the bindings they insert and the programs that update those bindings. A "*" means no occurrence. It appears that the binding "FindPersonDepend" in the "Person" environment is not updated. This indicates a potential error (the SPASM constraint 4b in Section 6.3 is violated), since the author of the personnel system used here as an example is expected to follow the persistent location binding methodology (Section 6.2.1).

```
USE-PROGRAM                BINDING                       STORED
--------------------------------------------------------------------------------
...                        ...                           ...
Person/ModifyPerson.N      Person\FindPersonDepend       Person\FindPersonDepend
Person/ModifyPerson.N      Person\GetInstancePerson      *
Person/ModifyPerson.N      Person\KeyChoosePerson        Person\KeyChoosePerson
Person/ModifyPerson.N      Person\ModifyPerson           Person\ModifyPerson
Person/ModifyPerson.N      Person\PersonKey              Person\PersonKey
Person/PersonValues2.N     Person\DuplicatePerson        Person\DuplicatePerson
Person/PersonValues2.N     Person\ShowPerson             Person\ShowPerson
Person/PersonValues2.N     Person\ShowPersons            Person\ShowPersons
*                          *                             Person\WritePersonName
...                        ...                           ...
```

Table 7.3: Use-stored dependency table

```
*****************************************************
*                                                   *
*                 *** ENVMAKE ***                   *
*                                                   *
*    H - Help                                       *
*                                                   *
*    A - Write program names with category          *
*                                                   *
*    I - Write insert-programs with bindings        *
*    U - Write update-programs with bindings        *
*    D - Write drop-programs with bindings          *
*    S - Write startup-programs with bindings       *
*    P - Write stored bindings                      *
*                                                   *
*   IP - Write insert/update dependency table       *
*   IU - Write insert/use dependency table          *
*   ID - Write insert/drop dependency table         *
*   IS - Write insert/stored dependency table       *
*   UU - Write update/use dependency table          *
*   UI - Write use/insert dependency table          *
*   US - Write use/stored dependency table          *
*   DU - Write drop/use dependency table            *
*   DS - Write drop/stored dependency table         *
*   TT - Write type_def/type_use dependency table   *
*                                                   *
*   PE - Write prog/env/binding matrix              *
*   EP - Write env/prog/binding matrix              *
*                                                   *
*   PS - Write program status                       *
*   TE - Write type databases                       *
*                                                   *
*   SI - Topological sort - insert-programs         *
*   TI - Topological sort - type-programs           *
*                                                   *
*    E - Exit (Back to NPE main menu)               *
*                                                   *
*****************************************************
```

Figure 7.1: The EnvMake menu

Table 7.3 is another kind of dependency table. It shows the bindings (second column) occurring in the use-clauses of the programs in the first column. The third column shows the corresponding bindings that are actually in the persistent store (at the time of the last thesaurus update). A "*" in each of the two leftmost columns indicates that no program is using the binding found in the store (in this case "WritePersonName" in the "Person" environment). This is a violation of the SPASM constraint 8c. According to Table 6.7, "WritePersonName" should either be used in some program or be dropped from the persistent store.

As shown in the menu of Figure 7.1, EnvMake offers several kinds of dependency tables. EnvMake can also visualise dependency information in the form of matrices showing which operations are performed on which bindings in which programs. Table 7.4 shows the names of the bindings inserted, used or dropped in two programs. The table is an excerpt from the full table generated as a result of selecting the "PE" option in the EnvMake menu. A similar table sorted by environments rather than programs is generated when selecting the "EP" option.

```
PROGRAM         ENVIRONMENT      INSERTED      USED                DROPPED
------------------------------------------------------------------------------
DropCompany.N   Company_Org      *             EDITIONs            EDITIONs
                                 *             LIBRARIEs           LIBRARIEs
                                 *             PERSONs             PERSONs
                                 *             PRODUCTs            PRODUCTs
                                 *             PROJECTs            PROJECTs
                                 *             TASKs               TASKs
                                 *             TEAMs               TEAMs
                                 *             VERSIONs            VERSIONs
                PS               *             User                *
                User             *             Company_Org         *

DeletePerson.N  Company_Org      *             PERSONs             *
                                 *             PROJECTs            *
                                 *             TEAMs               *
                GlasgowLibraries *             Lists               *
                IO               *             writeString         *
                Lists            *             cons                *
                                 *             hd                  *
                                 *             l_empty             *
                                 *             l_isu_append        *
                                 *             tl                  *
                PS               *             GlasgowLibraries    *
                                 *             IO                  *
                                 *             User                *
                Person           *             DeletePerson        *
                                 *             DeletePersonReferenc*
                                 *             DuplicatePerson     *
                                 *             KeyChoosePerson     *
                                 *             MakeDummyProject    *
                User             *             Company_Org         *
                                 *             Person              *
...             ...              ...           ...                 ...
------------------------------------------------------------------------------
Number of entries: 399
------------------------------------------------------------------------------
```

Table 7.4: Excerpt from a program-environment matrix

The information returned may be massive for large applications. The matrix of Table 7.4 originally had 399 entries, for example. Programmers have three options for restricting the output. They can select:

- kind of binding (procedure, structure, etc.);[1]
- a particular program, environment or binding by specifying a (sub)string; or
- bindings in internal environments only (i.e., excluding the standard libraries and other libraries).

The kind of the bindings and other information have deliberately been omitted from the tables and matrices to suppress details that would have obscured the overall structure. Programmers are advised to use the TSIT interfaces to obtain additional information.

The "TI" and "SI" options of the menu in Figure 7.1 initiate topological sorting on the programs. The names of the programs and bindings involved in a possible cycle will be printed. This issue is discussed further in Section 7.3.

The "PS" option writes the names, the time of the last compilation and the time of the last execution of all the programs registered with a given application. The "TE" option provides information about which type databases are being used in the application. These two options concern build management and will be discussed in Section 7.4.

In order to enable installation (see Section 7.4.4), there must exist a partial order among the insert-programs so that a binding inserted by one program can be executed before the program using that binding. In particular, an environment must be created before it can be populated. Similarly, to enable compilation, there must exist a partial order among type-programs. Determining an order among dependent type-programs may be a non-trivial task if there are several type-programs with dependencies between them (see Section 7.4.2). The "SI" and "TI" options of the EnvMake menu suggest a linear sequence compliant with the partial order if possible. If a loop exists, the name of the programs constituting the loop and the involved bindings (type definitions) are presented to the programmer in a table.

7.3 Supporting the SPASM Model

A feature of EnvMake is that it checks the SPASM constraints (Table 6.1) in the context of Napier88. The following sections discuss how the constraints are checked, how rigid the tool is and some programmer experiences.

One should note that consistency checking based on static analysis may not be complete for all kinds of program. In particular, some programs may apply a hidden operation, e.g. they execute a procedure variable or parameter defined elsewhere, or

1 See Section 5.2.

operations may occur within conditional constructs. Nevertheless, structurally obscure programs that invalidate checks may arise negligibly often (Section 7.5.2), and even for these programs tools like EnvMake may be useful. See further discussion in Section 7.5.2.

7.3.1 Checking the SPASM Constraints

The thesaurus information collected by TSIT enables EnvMake to automatically check the SPASM constraints.[1] Invoking EnvMake with the `consistency` parameter initiates a check of all the SPASM constraints. In the case of violation, EnvMake gives a warning and indicates the source of the violation. Only a warning is given since a violation is not necessarily an error but may be an anomaly indicating a situation that is liable to errors. If any program has been changed after the last thesaurus update, EnvMake warns that the program should be analysed with TSIT to ensure up-to-date consistency analysis (Section 7.5.1). The checks of most of the SPASM constraints are briefly described below.

Program Categories

EnvMake keeps track of the category of a program and gives a warning if it detects a program belonging to more than one of the categories. For example, a program with both an insert-declaration and a drop-clause would be categorised as both an insert-program and a drop-program. Programs violating this requirement are listed together with the name of their assigned categories.

Type Definitions

EnvMake writes a list of all type definitions and components of type definitions that are not used in any program. In the case of several type definitions with the same name, EnvMake informs about all the places in which these type definitions are used.

Declaration and Use

An insert/use dependency table similar to those described in Section 7.2 (Tables 7.2 and 7.3) forms the basis for the check of constraints 3a and 3b. Entries in the table corresponding to a violation are presented to the programmer (see analogous example in "Stub Constraints" below). EnvMake writes a list of all identifiers declared but not used, as specified in the constraints 3c, 3d and 3e.

[1] Information about the binding categories (Section 6.2.3) must be provided by the programmers in the current implementation. The default is that all bindings are regarded as internal (except those in the standard library). Some of the constraints would have been too restrictive if libraries and other exported or imported bindings were regarded as internal, and corresponding violation messages would have been felt inappropriate. Constraint 5a (Section 6.3) is meaningless without the categorisation.

Stub Constraints

EnvMake writes the names of variable procedures not declared as stubs (constraint 4a). In the case of attempting to update a stub in more than one program (constraint 4b), EnvMake writes a table informing the name of the binding and the names of the corresponding insert- and update-programs. Table 7.5 shows an example where two bindings are updated twice. Table 7.2 in Section 7.2 already showed an example of violation of constraint 4b in which a procedure was never updated.

```
CHECKING CONSTRAINT 4B: MORE THAN ONE UPDATE PROGRAM
------------------------------------------------------------------------------
INSERT PROGRAM                   BINDING                 UPDATE-PROGRAM
------------------------------------------------------------------------------
Person/DeletePerson_stub.N       Person/DeletePerson     Library/DeleteLibrary.N
Person/DeletePerson_stub.N       Person/DeletePerson     Person/DeletePerson.N
Person/DeletePersonRef_stub.N    Person/DeletePersonRef  Library/DeleteLibrary.N
Person/DeletePersonRef_stub.N    Person/DeletePersonRef  Person/DeletePerson.N
------------------------------------------------------------------------------
Number of entries: 4
------------------------------------------------------------------------------
```

Table 7.5: Insert-update dependency table

Drop-Clauses

The constraints involving drop-clauses are checked by using insert/drop dependency tables, and the results are presented in a form similar to Table 7.5.

Ordering of Insert- and Type-Programs

The check for partial order among the insert-programs and among the type-programs gives an error message if no such order exists and in that case presents a table of the programs in the loop and the bindings or type definitions involved.

Structuring and Naming Conventions

Discrepancies from the structuring conventions are reported as a list of the names of the directories with no corresponding environments (and vice versa). Names of environments, directories and files not following the naming scheme are also listed.

Persistent Store

The checks of the constraints in the eighth group detect inconsistencies between the source code and the actual contents of the persistent store. The use/stored table in Section 7.2 (Table 7.3) showed one example of an inconsistency (violating constraint 8c). The insert/stored and drop/stored dependency tables are also used to check the constraints in this group. If EnvMake has detected bindings in the store that are not used in any program, a future option of the tool could be that it (reflexively) generates and executes a corresponding drop-program upon user request. The user must be consulted because it could be the case that the binding would be used in some program under development.

7.3.2 Flexibility of EnvMake

The principle of EnvMake of giving warnings when violations of the SPASM constraints have been detected, can be compared with the way modern grammar checkers work (e.g. the checker in Microsoft™ Word Version 5.0). They check the text against some internal rules and give a warning if the text is not compliant with those rules. Then it is up to the programmer to resolve the problem: leave the text as it is or modify it (suggestions are usually provided).

EnvMake features optional selection of the constraints; programmers may "switch off" the check of individual constraints (cf. grammar checkers in which you can ignore rules). For example, a programmer may know that certain constraints will not be adhered to during a certain period of the development (typically during initial construction) and may wish to avoid the noise of unnecessary inconsistency messages.

Even though EnvMake supports and encourages the use of a certain programming methodology, it does not restrict its use to only systems that have been constructed in accordance with the methodology. Most of the constraints are useful whatever the methodology. For example, old software can be registered with EnvMake and make use of the facilities provided. Moreover, EnvMake does not fall over if violations are detected; it informs the programmer about the kind and source of violation and then checks the next constraint.

7.3.3 User Experiences

Parts of EnvMake have been successfully used. The implemented checks of the SPASM constraints were applied to Napier88 programs developed by seven programmers. Some of the experiences are described below.

- People claimed that the analysis increased their general understanding of their software.
- EnvMake assisted in providing a consistent naming structure; several had forgotten which names they had been using.
- The checks of unused identifiers stopped people from copying large segments of use-declarations from other programs without selecting only those needed. One person would still continue to do "bulk" copying: this was the simplest way, he said, and it was of no concern that some declarations were unused.
- One person had two cases in which two different programs updated the L-value of the same procedure in the persistent store and was very pleased to be informed about this error (see Table 7.5).
- People asked for new reports and were curious about the quality of their software compared with other people's software.

- Cases were discovered in which variables were declared in the store with stubs and then updated and used within the same program, without being used in any other program. Also other cases were detected in which global variables could be replaced by local variables.
- The various checks were useful for the author during the development of TSIT since a large piece of software developed by others had to be modified (the NinN compiler). For example, detecting procedure parameters not used within the procedure body enabled simplification of the procedure interface.[1]

The experiences described above indicates that EnvMake is useful. Nevertheless, a thorough experiment of studying the effect of using EnvMake (and TSIT) is being planned (Section 8.2.7). In additional to anecdotal information such as that described above, the extent to which programmers change their behaviour will be studied by comparing measurements of their software collected by TSIT before and after they adopt EnvMake. New requirements of EnvMake will also be an issue of such a study.

7.4 Build Management

A major feature of EnvMake is its support for build management (Section 3.3.2). In this context build management includes recompilation, re-execution (in compliance with the persistent location binding methodology) and installation of a release of a PAS. In particular, the following tasks must be carried out:

- All new and all changed programs should be compiled.
- When type-programs are compiled, the corresponding type databases should be updated accordingly.
- If a type-program is changed, all dependent programs should be compiled. The type-program must be compiled before the dependent programs. In particular, type-programs must be compiled in a correct order if there are dependencies among them.
- Programs that update persistent locations should be re-executed after change.
- Installation must be performed in a correct order, including all insert-programs before update-programs.

At present, these tasks are either performed in an *ad hoc* way or by use of Unix Makefiles. Programmers must infer dependencies and maintain Makefiles manually. EnvMake does not have any notion of "EnvMakefile" like Makefile in Make. All the necessary information is automatically inferred from the thesaurus and the internal data

1 Unfortunately, TSIT and EnvMake could be applied to themselves only at the end of their development – when they were being improved and tidied up – since they did not exist before they were developed, of course.

structures (e.g. dependency tables) of EnvMake. There are many advantages of such automation, and in particular it simplifies problems of change management as described in [Schwanke and Platoff 1989]:

> Many projects have a "catch-all" file of widely-used declarations. Maintainers are unwilling to create a new file to contain a new declaration, because of the nuisance of changing Makefiles, notifying the configuration management team, and so on.

A Makefile can also be regarded as a form of documentation of dependencies within the application. However, the kind of information that can be read from Makefiles can be obtained via the interactive EnvMake menu (Figure 7.1) or by querying the thesaurus directly (Section 5.3). The build management tasks of EnvMake are invoked by the `envMake` command with a parameter corresponding to the actual task:[1]

- `envMake plan`
- `envMake compile`
- `envMake run`
- `envMake install`

These tasks are discussed separately in the following sections.

7.4.1 Showing Status Information

Applying one of the parameters `compile`, `run` or `install` to the `envMake` command may initiate compilations. EnvMake determines the programs to be compiled and the order. Invoking EnvMake with the `run` or `install` parameter may initiate executions, and EnvMake determines a corresponding execution plan. This information, together with the times for the last compilation or execution, is written as a table on the screen before the actual compilation and execution. The corresponding program categories are also specified. A presentation of such a plan without actually performing any compilation or execution is provided by the `envMake plan` command.

Table 7.6 shows an example from the implementation of EnvMake application itself. It appears that *EnvMake_types.N* is the first program to be compiled. Type-programs are never executed so there are no corresponding entries in the two *Execution* columns. There are two rows with missing entries in the *Compilation* columns showing that there is no need for recompilation. Nevertheless, the corresponding executable versions (*dependTable.out* and *envMakeMainMenu.out*) should be executed. The program *installGen.N* appears with the value "00/00/00 00:00" for the last compilation indicating

[1] The current EnvMake implementation assumes that there is only one PAS in a given persistent store. A more sophisticated version should allow several PASs, and EnvMake would have to provide a mechanism for indicating the PAS being the subject of the task.

that this is a new program. Drop-programs and startup-programs are never automatically executed in the current version of EnvMake.

```
Compilation                       Execution                          Category
---------------------------------------------------------------------------------
Source program   Last compilation Executable program  Last executed
---------------------------------------------------------------------------------
EnvMake_types.N  21/08/92 21:35                                      type-prog
stubInsert.N     21/08/92 21:39                                      insert-prog
bindingMaps.N    21/08/92 22:35   bindingMaps.out     21/08/92 22:37 update-prog
createMatrix.N   22/08/92 09:35   createMatrix.out    22/08/92 09:35 update-prog
                                  dependTable.out     22/08/92 09:36 update-prog
                                  envMakeMainMenu.out 22/08/92 09:40 update-prog
installGen.N     00/00/00 00:00   installGen.out      00/00/00 00:00 update-prog
LValueUpdate.N   21/08/92 21:37   LValueUpdate.out    21/08/92 21:39 update-prog
drop.N           21/08/92 21:39                                      drop-prog
callInstallGen.N 21/08/92 22:55                                      startup-prog
```

Table 7.6: Compilation and execution plan

During compilation and execution, the name of the actual program and possible error messages are written to the screen. A log summarises the course of events as illustrated in Table 7.7 which shows that *createMatrix.N* failed compilation. The ".out" version was therefore not executed.

```
Compilation                       Execution                          Category
---------------------------------------------------------------------------------
Source program   Compiled         Executable program  Executed
---------------------------------------------------------------------------------
EnvMake_types.N  23/08/92 08:35                                      type-prog
stubInsert.N     23/08/92 09:19                                      insert-prog
bindingMaps.N    23/08/92 09:25   bindingMaps.out     23/08/92 09:30 update-prog
createMatrix.N   Error!           createMatrix.out    No execution   update-prog
                                  dependTable.out     23/08/92 09:36 update-prog
                                  envMakeMainMenu.out 23/08/92 09:40 update-prog
installGen.N     23/08/92 09:35   installGen.out      23/08/92 10.39 update-prog
LValueUpdate.N   23/08/92 09:37   LValueUpdate.out    23/08/92 10.42 update-prog
drop.N           23/08/92 09:39                                      drop-prog
callInstallGen.N 23/08/92 09:40                                      startup-prog
```

Table 7.7: Log of compilations and executions

The kind of status information described above is a significant improvement compared with what, e.g., Make provides – which is nothing. The present version of EnvMake offers only a textual interface for input and output. Later versions could provide a more sophisticated user interface with a separate log window displaying a table similar to that of Table 7.7 but with the time information and messages inserted into the *Compiled* and *Executed* columns as the compilation and execution proceed. Colours could also be used – error messages in red, etc.

7.4.2 Compilation

Invoking EnvMake with one of the parameters `compile`, `run` and `install`, causes every new or changed program to be (re)compiled.[1] In addition, programs that depend on modified type-programs will also be recompiled. If no type-programs have been changed, the order of compilation is not significant. If a type-program has been modified, however, the collection of dependent programs (which may include other type-programs) must be determined. Type-programs must precede the dependent programs in the compilation order. If compilation of a type-program fails, EnvMake will not initiate compilation of the (unchanged) dependent programs since those compilations would also fail or be based on obsolete versions of the type definitions.

There are two problems concerned with "the collection of dependent programs" of a type-program. First, it is to define the semantics of *dependency* in this context; second, it is to detect the actual programs. A simple rule of dependency is: all programs in a PAS or a subsystem are dependent on all the type-programs in that PAS or subsystem.[2] That is, if a type-program is changed, then all programs are recompiled to ensure that no program refers to old type definitions.[3] Applying this rule causes many programs to be recompiled unnecessarily. This very simple way of defining dependency is mainly due to the lack of appropriate tools for determining the collection of dependent programs according to a more refined definition.[4]

Probably the most obvious and correct way of defining type dependency is to regard a program as dependent on the specific type definitions that it uses. So, if a type definition (say) *T* is used in a program (say) *prog1.N*, then *prog1.N* is said to be dependent on *T*. In order to preserve consistency, a removal of or change to *T* requires *prog1.N* to be changed accordingly (if necessary) and thereafter recompiled. In the current EnvMake implementation, however, no test has been implemented in order to detect a change to a specific type definition (though such a test could be implemented by appropriate type graph comparison algorithms [Connor 1991]).[5] Nonetheless, the type-program containing the changed type definition will be detected as changed by the conventional timestamping technique. Therefore, as a compromise for implementation reasons, changes are recorded at the granularity of type-programs rather than the level of

1 Changes are detected by the timestamping technique, as in Make.

2 This seems to be the commonly practised rule.

3 Some implementations of type processing will make type changes available to programs that use the types indirectly without the need for reprocessing these programs (e.g. relational databases that use type names as keys).

4 This problem of the coarse level of dependencies between a compilation context and the compilation units is described more generally in [Tichy 1986] (see Section 3.3.2.2).

5 In the implementation of a tool described in [Tichy 1986], changes to declarations are detected by comparing their identifiers and their respective abstract syntax trees.

type definitions. Hence, the type dependencies shown in Figure 7.2 indicate that at least one type definition in the program at the arrow tail is used in the program at the arrow head. For example, *prog1.N* and *B_types.N* depend on *A_types.N* implying that EnvMake would initiate a recompilation of *prog1.N* and *B_types.N* after any change to *A_types.N*.

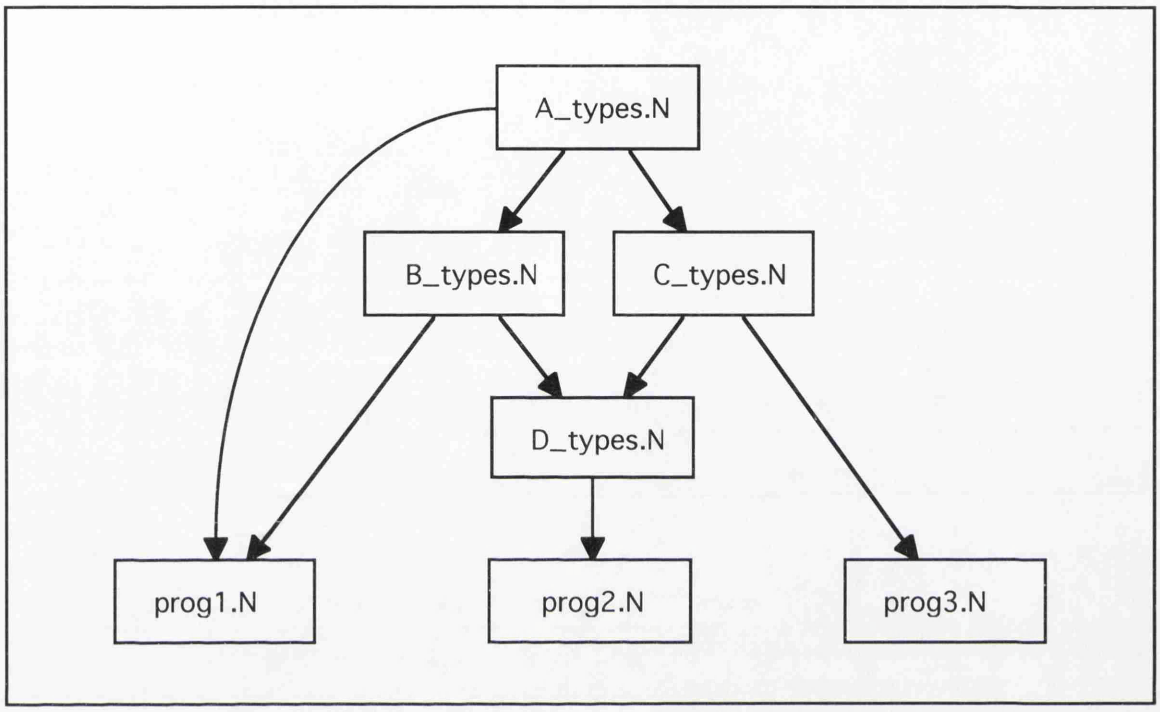


Figure 7.2: Type dependencies

In larger PASs it is common to have several type-programs, each of which covers a subsystem. For example, the WIN system [Cutts *et al.* 1990] has five type-programs; one in each of the subsystems "system", "lineEditor", "windowEditor", "editors" and "managers". By compiling with the `nps` command instead of the standard `npc`, the type definitions are inserted into a *type database* in the persistent store. Programs can be compiled later on against the definitions in such type databases (Section 4.2.1.1). EnvMake chooses the `nps` command when it compiles type-programs, ensuring that the corresponding type databases are up-to-date. Moreover, EnvMake automatically compiles against these type databases whenever it compiles a program within the corresponding subsystem.

As mentioned above, a type-program must be compiled before its dependent programs. The order is determined by the topological sort (Section 6.3.6). A change to a type definition does not only imply that the dependent programs need to be recompiled – they also generally need to be edited. Hence, these programs will be detected as changed by the timestamping algorithm and thus be subject of recompilation in any case. However, there are still at least two reasons for applying the type dependency rule discussed above. First, there are cases in which programs do not need editing but still need recompilation.[1] Second, if the dependent programs need editing, but this has not been done for some reasons, recompilation ensures that type inconsistencies are detected at an early stage rather than later at run-time (e.g. in use-clauses).

Avoiding unnecessary recompilations is not a new problem (Section 3.3.2.2). It could be argued that the significant increase in machine power would make this problem redundant. However, the compilation time may still be considerable, and the size of application systems is continuously increasing, so the desire to avoid total recompilation after minor changes will undoubtedly continue. As explained above, EnvMake alleviates this problem by recompiling only changed programs and those that depend on them.

7.4.3 Execution

When the `envMake` command is invoked with the `run` parameter, it is first checked for necessary compilations according to the process described in the previous section. Thereafter, EnvMake starts execution of update-programs that have been recently compiled. The order of execution is not significant. Keeping track of which programs that are update-programs and when they should be executed is time consuming and error-prone without appropriate tool support.

One may question when programs of the other categories are executed since they are not executed when `envMake run` is requested. As mentioned, type-programs are never executed. Insert-programs are executed as part of a system installation (next section) and maintenance (including type evolution). Startup-programs typically call persistent procedures to initiate a particular task or invoke the menu of an interactive application. These programs are executed on user request only. Drop-programs are normally created and executed *ad hoc* but could be automatically generated and executed by a more sophisticated version of EnvMake (Section 7.6).

7.4.4 Installation

When installing bindings into a persistent store, the installation-order is significant. That is, a binding must be inserted before it can be used. For example, an environment must be created before it can be populated, a location must be created for a procedure before its L-value can be updated, etc.[2] EnvMake provides automatic installation. When the files

1 For example, if a type definition of kind variant has been extended with a new branch, programs that should not reference this new branch do not need to be changed. For type equivalence purposes, however, they need to be recompiled. (There exist languages, for example Machiavelli [Ohori *et al.* 1989], where a partial match suffices, and for such languages recompilation would be unnecessary.)

2 The author experienced difficulties when installing a relatively large piece of software developed by others (a modified version of the NinN compiler). The original Makefiles could not be used since

of a PAS have been stored in the respective directories of the file system, the necessary installation into the persistent store is provided by `envMake install` which performs the following tasks:

i) compilation of type-programs (in topologically sorted order)

ii) compilation of other programs (in any order)

iii) execution of insert-programs (in topologically sorted order)

iv) execution of update-programs (in any order)

The type-programs are compiled (by the `nps` command) in a topologically sorted order. Then the rest of the programs are compiled (if needed) in any order. Only insert- and update-programs are executed in an installation.

7.5 Implementation

EnvMake is implemented in Napier88, and Maps [Atkinson *et al.* 1990] are heavily used. The tool is tightly integrated with TSIT (Chapter 5). All the internal data structures of EnvMake containing information about dependencies, program categories, timestamps, etc. are based on the thesaurus information. The internal EnvMake information is updated immediately after the TSIT program analysis and subsequent thesaurus update. The actual update of timestamp information is managed by EnvMake itself,[1] but the files being operated on are those registered with TSIT (Section 5.4). Hence, any addition, deletion or renaming of a file belonging to the actual PAS must be registered with TSIT before the change becomes visible in EnvMake.

In the current implementation the type databases are given the same names as the names of the PASs or subsystems, which can be extracted from the files holding the corresponding type-programs: ⟨PAS or subsys⟩_types.N.

7.5.1 Problems with Ensuring Up-To-Date Information

A potential problem pertains to EnvMake's build management feature of automatic program executions. Only programs of certain categories (insert-programs and update-programs) should be executed, and the order is significant (insert-programs before other programs, etc.). The order is determined on the basis of the thesaurus information, which provides the input to the topological sorting algorithm. If EnvMake has detected a file as being changed after inspecting the timestamp information, it *might* be the case that the change concerns code relating to insert, update or use of bindings which in turn may

they had not been updated in accordance with the changes to the code. The installation-order was eventually determined by trial-and-error.

1 The time information provided by the Unix file system is used to detect when a file was last changed.

affect the execution order. Theoretically, a program may also have changed its category (e.g. from startup-program to update-program) indicating that the program now should be automatically executed.[1] Such changes will not be detected before the next TSIT analysis and thesaurus update. Therefore, the strictly correct sequence of EnvMake's tasks would be as follows:

1) Inspect time stamps of the registered files.
2) Compile all new or changed programs.
3) If no compilation errors, let TSIT analyse the new or changed programs and update the thesaurus correspondingly.
4) Update dependency tables and program category information.
5) Perform topological sort.
6) Execute the programs according to the (possibly new) order.

A significant overhead is caused by the third step. So, due to performance reasons, that step is sacrificed (implying that also the fourth step is ignored since the thesaurus information is unchanged). This is reasonable since in practice very few changes affect the execution order, and change of program category is extremely rare. If the thesaurus is updated (say) every night, the analysed information is still relatively up-to-date. However, if a programmer knows that he or she changed code pertaining to insert, update or use of bindings, then a thesaurus update should be requested promptly.

Another problem pertains to the compilation order of type-programs which is determined on the basis of dependencies among type definitions. These dependencies can be inferred from the thesaurus information. This is too late, however, since the programs must already be fed into TSIT in a topologically sorted order. Like the compiler, TSIT must first analyse the program containing certain type definitions before it can analyse the programs that use these type definitions. To solve the problem, EnvMake is enhanced with a special procedure that analyses type-programs (EnvMake detects if they really are type-programs) with the only purpose of determining the order among them. A two-pass analyser does the job. In the first pass all the type definitions are found; in the second pass all the uses are found. A type dependency table is then generated which is taken as input by the topological sorting algorithm.

7.5.2 Problems of Naming and Identity

The current EnvMake implementation is name-based only. The SPASM constraints are checked under the assumption that environments are uniquely identified by their names

[1] Programmers should be encouraged *not* to change categories. Such an undisciplined practice makes maintenance difficult, particularly in large projects with many people involved (see Section 6.6).

and by paths of named environments from a persistent root if such paths exist. If there are no paths, an environment is assumed to be identified by its name only. It is also assumed that a given environment has only one name. The dependency tables (Section 7.2) on which the build management features of EnvMake are based, rely on these assumptions. Problems may occur in the following three cases:

i) Environments are returned by procedures.

ii) Vectors, structures, variants, procedures, etc. have environments as elements, fields, branches, parameters, etc.

iii) Different identifiers denote the same environment.

The misleading warnings that might be given in these cases can be compared to grammar checkers that suggest changes in complex but entirely correct sentences.[1]

7.5.2.1 Returned Environments

If a procedure returns an environment such as *mk_env* in p1.N[2] in Figure 7.3, then the environment name used within the procedure body (*e* in that example) generally differs from the name of the identifier being assigned that environment in a call to the procedure (*list* in p2.N).

```
p1.N
use PS() with lib : env;
              date : proc( -> string );
              environment : proc( -> env ) in
  in lib let mk_env = proc( -> env)
    begin
      let e = environment()
      in e let created = date()
      e
    end

p2.N
use PS() with lib : env in
use lib with mk_env : proc( -> env ) in
in lib let list = mk_env()

p3.N
use PS() with lib : env in
use lib with lists : env in
use lists with created : string in {...}
```

Figure 7.3: Environment as result type

1 The suggestion does not necessarily concern the fact that the sentence is complex, but the sentence is too complex for the grammar checker to parse it correctly.

2 This is an example from a real application (the maps library).

Moreover, there is no path associated with the environment in the procedure body. For example, checking that the binding *created* inserted into *e* is ever used is impossible since the binding is identified as *e\created* when it is inserted and as *PS()\lib\list\created* if it is used as an element in the *list* environment (p3.N).

EnvMake would in this case report that *e\created* was unused and *PS()\lib\list\created* was undeclared, so EnvMake would issue unnecessary warnings. Probably what would happen, however, is that a programmer would use TSIT to find all occurrences of *created* and would then resolve the problem.

Except for four standard procedures, there are only two procedures in the eight analysed applications that return environments, which indicates the problem is not that severe.

7.5.2.2 Environments in other Data Structures

As shown in Table 5.11, if an element of a vector is accessed, only the vector name is registered in the thesaurus. This is reasonable since a vector element does not have a name but is identified by an index. If a structure, variant or procedure has environments as fields, branches or parameters, then the environments are "identified" by the name of the field, branch or parameter in addition to the name of the associated structure, variant or procedure. However, this does not ensure globally unique naming and may compromise the quality of the EnvMake support. Nevertheless, in most cases the names *are* unique within an application.

env	BindingInserted	372
env	ProcParamDecl	103
env	StructFieldDecl	1
env	ValueDecl	388
env	Total	864

Table 7.8: Declaration of environments[1]

Among the 864 declarations[2] of environments in the analysed applications, there are 103 cases where environments are procedure parameters, one case with an environment as a structure field and no cases where environments are variant branches or vector elements (Table 7.8). There are 372 cases in which environments are declared directly[3] into other environments (usually persistent), and 388 cases in which they are declared directly but not into other environments.

1 This is an excerpt from a large table in [Sjøberg 1992].

2 Use-clause declarations are excluded.

3 That is, not as part of another structure (except environments).

7.5.2.3 Aliases to Environments

A potential problem occurs if several identifiers in a program or in the persistent store denote the same environment. For example, in the following code a variable *x* is declared in the environment *e1*. Then *e1* is being assigned to another environment *e2*, so that *e1* and *e2* denote the same environment. Finally, *x* occurs in a use-clause of *e2*.

```
in e1 let x := 2
    e2 := e1
use e2 with x : int in ...
```

EnvMake would at present claim (incorrectly) that the variable *x* in the environment *e1* (identified as *e1\x*) is not used and that *x* in *e2* (identified as *e2\x*) is not declared. To solve this problem, some kinds of alias list could be constructed by sophisticated source code analysis in a future version of EnvMake or TSIT.[1] Also, since the thesaurus itself is located in the persistent store, it could be enhanced in that its entries could contain references to the bindings themselves (rather than only containing their names).[2] This might enable the thesaurus to store information about different identifiers referring to the same environment, and thus a list of aliases could be constructed for each environment. EnvMake could in turn use this information to improve the quality of its analyses. The need for such alias lists, however, does not seem especially pressing. There are only three assignments involving environments (0.1% of all assignments) in the analysed applications.

7.6 Future Development of EnvMake

The SPASM constraint that all programs should belong to exactly one of the five categories described in Section 6.2.2 makes *individual* programs easier to understand (and thus to write, update, manage, etc.). On the other hand, there will be many more programs, which leads to it being potentially harder to comprehend the total system. Supporting tools for program management (automatic generation, compilation and execution) are therefore essential.

This section describes a tentative proposal (i.e., no implementation and no evaluation) for how EnvMake can be enhanced to offer support for the persistent location binding methodology when also the type of a location is changed (not only the contents). For example, if a procedure type is changed, the same location cannot be used any more. It is necessary to drop the old location and create a new location with a new procedure.

Figure 6.9 in Section 6.5.1 summarises the necessary steps to be carried out in compliance with a certain strategy involving four files per binding. The steps presented

1 Hyper-programming (Section 8.2.6) may finesse this problem.

2 See also Section 8.2.5.

below are compliant with that strategy. For each step the perpetrator is indicated (EnvMake or the programmer). So, if the type of a binding is to be changed, the following steps should be carried out:

1) EnvMake drops the existing location by executing the program in ⟨binding⟩_drop.N.
2) EnvMake removes the old ⟨binding⟩_insert.N.
3) User edits ⟨binding⟩_update.N, and EnvMake infers the new type by analysing that program.[1]
4) On the basis of the information obtained in step (3), EnvMake generates a new ⟨binding⟩_insert.N with a stub. If the binding to be created is a procedure, then the stub includes a call to the *uninitialised* or *uninitialised_void* procedure (for security and debugging purposes)[2] depending on whether or not the procedure returns a result. Then EnvMake compiles and executes ⟨binding⟩_insert.N.
5) EnvMake recompiles and re-executes ⟨binding⟩_update.N.
6) EnvMake presents information about all the programs that use the binding, indicating which must be edited by the programmer. EnvMake (re)compiles and re-executes as required.

If a new binding is to be created from scratch, steps (1), (2) and (6) should be ignored.

EnvMake can also be enhanced to support other aspects of construction and maintenance, such as organising directories and environments. For example, assume that a programmer manually creates the root directory of a PAS and (recursively) all subdirectories.[3] The path of the root directory could then be passed to EnvMake which in turn could construct a matching hierarchy of environments in the corresponding ⟨PAS⟩ environment (or vice versa). In this way EnvMake would ensure isomorphism both in structure and naming (cf. SPASM constraint 7a).

A whole class of tools that would enhance the EnvMake programming environment can be envisaged; some examples follow.

- EnvMake could automatically generate use-clauses.
- EnvMake could optimise programs to convert **use** paths into hyper-references and do the inverse to prepare code for shipment.
- EnvMake could reflexively generate, compile and execute programs that fetch remote libraries when they are used the first time.

1 Alternatively, the programmer specifies the environment path, name and the new type interactively in a dialogue with EnvMake.

2 See Section 6.2.1.

3 A possible enhancement is that the programmer specifies all the directories in a shorthand notation or in a dialogue and leaves the actual creation to EnvMake.

- EnvMake could store annotations in environments to improve presentation and to support various forms of automation.

The first two proposals are discussed further in Section 8.2.

7.7 Summary

This chapter has described a tool called EnvMake which supports application construction and maintenance in a persistent programming environment. Even though the EnvMake has been developed in the context of Napier88, the principles behind the tool apply to all persistent programming languages providing higher-order procedures and L-value binding to persistent locations.

EnvMake has been tailored to support the programming methodology and the SPASM model described in the previous chapter. For each violation of a SPASM constraint, EnvMake produces a warning message and an indication of the source of the violation. It is then the responsibility of the programmer to rectify the inconsistent state. (An enhanced version of EnvMake could often offer a solution.) Several of the SPASM constraints prevent situations liable to provoke run-time errors. The corresponding EnvMake checks are performed at "EnvMake-time" (which is between compile-time and run-time)[1] and, as such, comply with the principle of "eager checking" [Atkinson *et al.* 1988], i.e., performing as much as possible of the checking as early as possible.

As part of the persistent location binding methodology, some programs should only be recompiled after change, some should also be re-executed (those that update code in the persistent store), programs with type definitions should update the corresponding type databases when they are compiled, etc. Traditionally, programmers have carried out these tasks manually, or they have created and manually maintained Unix Makefiles for the tasks. Both these strategies are tedious and error-prone, especially for large PASs. On the basis of the thesaurus contents, EnvMake infers which programs should be recompiled, which ones should also be re-executed, etc. This information, together with timestamp information about the last change and compilation of a program, enables EnvMake to automatically perform all the needed recompilations and re-executions.

The build management features of EnvMake support incremental update of programs stored in persistent locations as long as the types of the locations are unchanged. At the end of this chapter, it was proposed how EnvMake could automate most of the tasks needed to change the location type as well.

Automatic build management tools have also been developed in other programming environments, mostly for C (e.g. THINK C™ for the Macintosh [Symantec 1989]).

[1] EnvMake is based on the thesaurus information which in turn is extracted from source programs after they have been checked for compilation errors (Section 5.4).

However, persistent programming environments are generally more sophisticated in that they include issues that were traditionally dealt with by the operating system or DBMS, so they require correspondingly more sophisticated build management tools, such as EnvMake.

EnvMake imposes a certain view on the programming process. There is a trade-off between support and flexibility. The more a programmer complies with the model of EnvMake, the more assistance EnvMake provides. The conventions and constraints imposed by EnvMake should not be regarded as a hindrance. On the contrary, they encourage the use of the persistent store in a disciplined way and assist in providing a common model for construction and maintenance of PASs in a community of software builders.

Chapter 8

Conclusions and Future Work

One of the most challenging problems of building and maintaining large, long-lived data-intensive application systems is to cope with all the changes that inevitably will be imposed on the systems over time. The motivation for the research presented in this thesis is to simplify and aid the process of changing such systems by providing supporting models, methodologies and tools. The thesis has demonstrated that automatically generated thesauri prove a suitable basis for achievements in that direction.

It is sometimes argued that change is a consequence of poor design or erroneous implementation. Naturally, some changes arise from these causes, and improved techniques for reducing them are valid research. However, the major cause of change is perceived to be user initiated, and the thesis takes the view that it is important to facilitate such change so that the people using a persistent application system are not discouraged from innovation.

Most of the research was conducted in the context of the strongly typed, persistent programming language Napier88. The ideas behind the introduced SPASM model, the methodologies and the EnvMake tool, however, are independent of Napier88 and can thus be applied to any persistent or database programming environment (e.g. persistent object-oriented systems) in which programs and other data reside in a persistent store.

8.1 Summary – Utilisation of Thesauri

The basis for the work presented in this thesis is automatically created and maintained thesauri which contain extensive information about all names used in the implementations of persistent application systems. Our understanding of a system is closely related to the use of names. Names are chosen as a focus on the assumption that most of the time they will be used consistently by people throughout the life of a system.

8.1.1 Quantifying Evolution

It is commonly known that there is a significant number of changes going on in the application software industry, but the kind and scale of various forms of change should now be quantified. This thesis has introduced a research direction concerning the problem of quantifying schema evolution. A relational database application, a health management system, was studied in depth during an 18 month period. The study reveals that schema changes are significant both in the development period *and* after the system has become operational. The main results were:

- Number of relations: 139% increase.
- Number of fields: 274% increase.
- Every relation was changed.
- 35% more additions than deletions.

The consequences of the schema changes on the application programs have also been measured. The results confirm the need for change management tools.

The study reported has wider applicability than just to traditional database systems. The data descriptions and consequently dependent data (including programs) of all persistent application systems will inevitably have to be changed in order to reflect the changing user needs. That is, schema evolution in traditional databases corresponds to class evolution in object-oriented database systems, to type evolution in applications developed in strongly typed, persistent programming languages (e.g. Napier88) and, at a higher level, to changes to application models described in the framework of conceptual data models (e.g. the Entity-Relationship model).

The measurements were obtained by the HMS thesaurus tool which analyses the database schemata and application programs. The tool spans all the languages used to build the whole persistent application system, its user interfaces and its databases. Information about programmer-introduced names denoting relations, fields, screens, actions, queries, update functions, etc. is extracted and inserted into the thesaurus. Changes to the set of occurrences of these names are also recorded. In particular, the tool provides information about how many screens, actions, queries, etc. may be affected by a potential schema change and can thus be used to estimate the costs of this change. Some of the statistics presented and the thesaurus' raw data reveal possibilities concerning optimisation strategies.

We have only been able to find one report on similar measurements [Marche 1993]. In that case, change[1] was measured at the data modelling level. Also this study confirms the significant extent of change. In general, change statistics from other projects should be collected, enabling systems in various application domains to be compared in a larger, more representative study.

The causes of change may also vary from system to system.[2] These causes, however, are another research issue and are regarded as irrelevant in our context. The key point is that our measurements of a real, industrial system confirm that designers of tools for the management of large, long-lived systems involving databases must address the problem of changes to schemata. The traditional view of first defining a (fixed) schema and thereafter developing the dependent application programs has proved inappropriate.

8.1.2 Thesauri in a Strongly Typed Persistent Environment

Persistent languages potentially support construction and maintenance of long-lived, data-intensive, application systems. To exploit the benefits of persistence, however, supporting models, methodologies and tools must be developed. Automatically generated thesauri are a suitable platform for such development.

The Thesaurus-based Software Information Tool (TSIT), based on the same ideas as the HMS thesaurus tool, was implemented for and in the strongly typed persistent language Napier88. The heart of TSIT is the thesaurus which keeps track of identifiers of all kinds used in the application. Information such as type, container, context, declaration/use, etc. is recorded for each identifier occurrence. TSIT provides impact analysis (consequences of change) and a simple query interface (Figure 8.1). TSIT can also be used as a tool to generate measurements of various kinds. In particular, to support the arguments of the thesis, eight Napier88 applications were measured in detail. In total, 51328 lines of code with 84501 name occurrences in 367 programs were analysed. The measurements comprise the use of names, the use of various language constructs, the extent of inconsistencies, how programs interact with environments in the persistent store and other programming issues. Some specific measurements were undertaken in response to questions about language usage from language designers at the University of St Andrews.

1 No measurements on *consequences* of change were reported.

2 In the HMS case considerable investment (much in excess of coding costs) went into design and planning. Changes were still encountered due to changing organisational needs, changing regulations and the addition of major new subsystems.

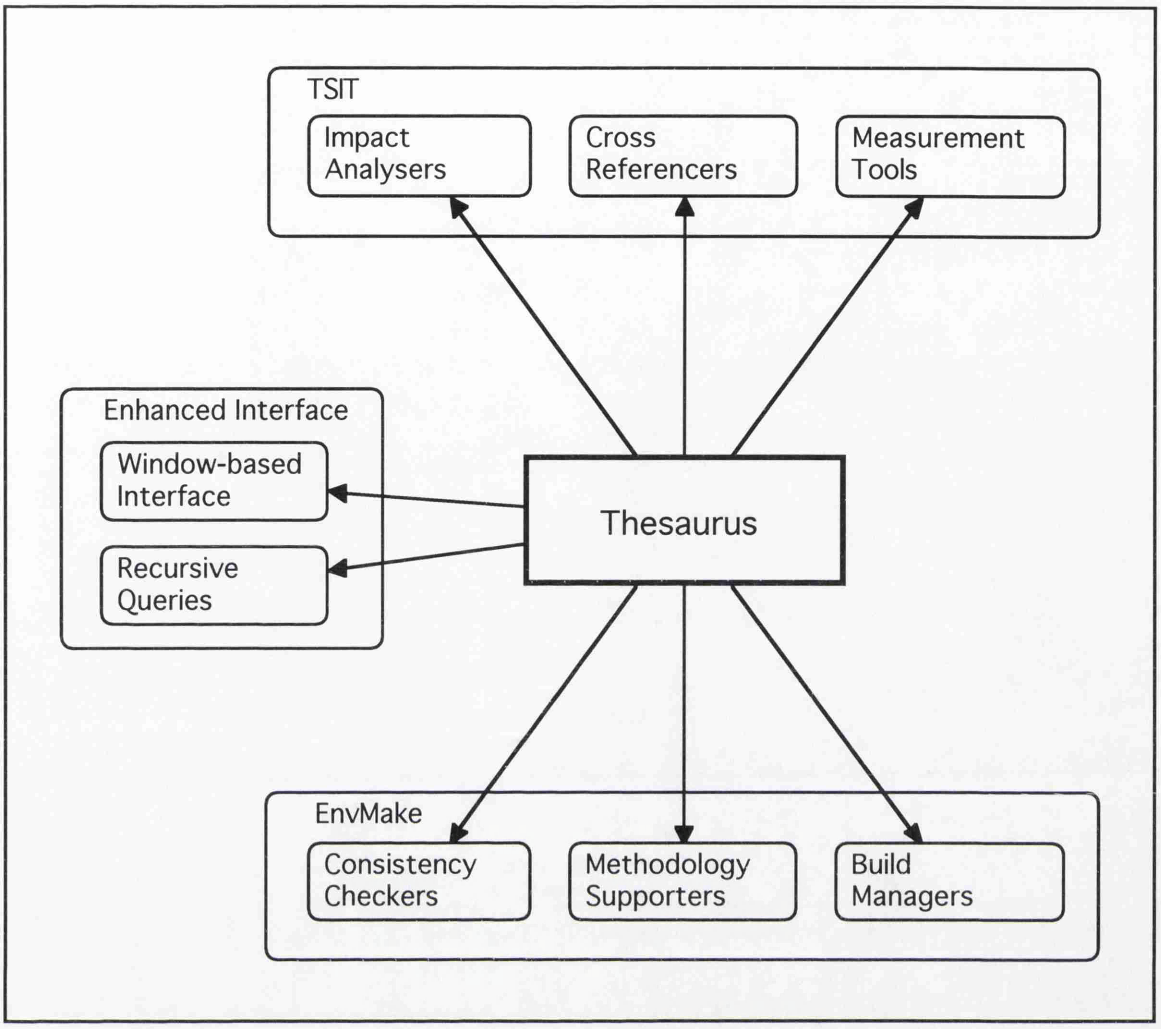


Figure 8.1: Thesaurus-based tools

Models and methodologies for persistent software development are still in their infancy, but there are already many activities that are suitable for automation or that would benefit from supporting tools. One example of such a tool is EnvMake which also utilises the thesaurus information (Figure 8.1).

In general, the use of persistence has made it easy to build programs and applications working on top of the thesaurus. Two examples are the enhanced interfaces to the thesaurus – a sophisticated query language [Trinder 1991] and a window-based, menu-driven interface [Sjøberg *et al.* 1993].

The detailed information about application programs and data provided by the thesaurus, and its user interfaces, enable software builders and maintainers to explore and extract information of particular interest. By investing in creating tailored interfaces, they can study evolution and other problems of application construction and maintenance from their specific points of view.

8.1.3 Models and Methodologies

Comparing file-based program construction methodologies with those based on persistent stores, we observe that in the persistent store case all possible data structures and their types are accommodated and preserved when data is stored for later use or passed between programs. Typically, file-based program construction has little support from the type system and perforce loses structural information as data is mapped to a sequence of bytes. Although persistence leads to more sophisticated interfaces between program parts using arbitrary modules, we believe it will ultimately yield benefits because of the significant structural information that is conveyed between programs. Thesaurus tools will be better able to infer dependencies and verify model constraints by analysis of this richer structural information. However, to fully benefit from the new technology, comprehensive programming methodologies are needed.

This thesis takes a further step in that direction; it has introduced a construction and maintenance methodology together with a structured persistent application system model (SPASM) that specifies an architecture for application systems developed in Napier88. SPASM defines a set of constraints with which each suite of application software should comply. At the time of writing, there are 24 constraints like the following: "a binding inserted into the store, not intended for export, should be used somewhere within the application", "all type definitions should be used within the application", "there should be exactly one program updating a procedure (or some other kinds of value) bound to a persistent location initialised with a stub", etc. A violation of a constraint could be a logical error, or it may just indicate a situation that might eventually cause problems. Inconsistent states will be the normal case, particularly during the initial development. Programmers may find it helpful to be able to request that certain subsets of these inconsistencies be enumerated.

Both SPASM and the methodology are general in that they are independent of the applications being implemented. They are, however, couched in terms of the programming language (Napier88) even though most of the principles they encode are applicable to any persistent or database programming environment.

Methodologies and constraints could be felt as a burden by some programmers since they may already have adopted their own more or less good programming style. Nevertheless, in the business of large-scale software application development, with typically many people on the same project, it is crucial that people work in a disciplined way. Models and methodologies should be perceived as supportive rather than restrictive if they are based on well-founded principles and the common experiences of several programmers. The availability of supporting tools may, however, influence the choice of methodologies. Some approaches may be very convenient if there are corresponding tools but infeasible if there are not. The next section describes EnvMake – an example of such a tool.

8.1.4 EnvMake

EnvMake is a thesaurus-based tool that supports persistent programmers in the process of creating and maintaining large application systems. The provision of persistence, enabling applications and tools to be contained in the same store and implemented in the same language, creates new possibilities for enhanced and more integrated CASE tools. EnvMake is a demonstration of what such tools could be like.

There are many tools available to support application development using file-based code, e.g. *RCS, Make, awk, grep,* etc. Analogous tools are required to operate on code in persistent stores. Potentially these tools can be superior to those operating on byte-stream files because a persistent store is coherent, transactional, structured and typed. Programs are no longer large discrete units; instead they are smaller and typically extract and use subprograms from the persistent store. Under this model libraries are potentially easy to use, and large applications can be constructed incrementally. Code reuse could also be simplified with suitable tools.

The information required by EnvMake is generated automatically. Most of the needed information is obtained directly from the thesaurus, but EnvMake also holds some internal data structures, e.g. for keeping track of time stamps required for determining necessary recompilation and re-execution. The present features of EnvMake include visualisation of structures and dependencies of an application, methodology support, checking model adherence and incremental build management.

8.1.4.1 Structure and Dependency Visualisation

EnvMake provides programmers and other parts of EnvMake itself (the SPASM checking and build management components, see below) with a table showing dependencies between programs that insert a binding and those that update the binding – a so-called "insert/update dependency table". There are similar dependency tables for insert/use, update/use, drop/stored, etc. Another form of visualisation is matrices showing which programs perform which operations on which environments in the persistent store.

8.1.4.2 Supporting Steps of the Construction and Maintenance Methodology

EnvMake has been designed to support a strategy for implementing the persistent location binding methodology. EnvMake automatically generates the programs that insert or drop a binding. The user only needs to create or edit the program that updates the location with a meaningful value. In addition, if the type of the binding changes, the user must change the programs using the binding. These programs are indicated by EnvMake.

EnvMake also assists in other aspects of construction and maintenance such as organising the structure of environments in the persistent store and directories in the file system. EnvMake ensures isomorphism and adherence to naming conventions by actively taking part in the creation and maintenance of files and environments.

8.1.4.3 Checking the SPASM Constraints

The compiler of a programming language already performs many forms of consistency checks within a program such as type checking, ensuring declaration and unique naming of identifiers, etc. EnvMake is concerned with complementary checks such as those between programs and those between programs and bindings in the persistent store. Being specific, EnvMake checks the 24 SPASM constraints.[1]

Several of the constraints are based on a categorisation of programs according to their semantics. On the criteria of how they operate on the persistent store and where types are defined the programs are divided into the following categories:

- *Type-program* – a program whose contents are exclusively type definitions.
- *Insert-program* – a program that inserts at least one binding but neither updates a persistent location nor drops any binding.
- *Update-program* – a program that updates at least one persistent location but neither inserts nor drops any binding.
- *Drop-program* – a program that drops at least one binding but neither updates a persistent location nor inserts any binding.
- *Startup-program* – a program that uses at least one binding but neither updates a persistent location, inserts nor drops any binding.

This categorisation, which is done automatically by EnvMake, is also the basis for the build management features described below.

8.1.4.4 Build Management

At present, many Napier88 programmers use Make to install software and to help rebuild applications after change. When using Make, the programmers have to manually work out the order of installing components into the persistent store. This may be a difficult task for non-trivial applications. A component must be inserted into the store before it can be used by another component. EnvMake determines the correct installation order by topological sorting and initiates execution of the respective insert-programs.

Moreover, when using Make, the programmers also have to manually specify compilation and execution dependencies such as the following:

1 At the time of writing, some of the checks still have to be implemented.

- if a type-program is changed, all dependent programs should be compiled;
- all type-programs must be compiled before other (dependent) programs;
- type-programs must be compiled in correct order if there are dependencies among them;
- the type database associated with a type-program should be updated;
- a binding must be inserted before it is referred to in another insert-program (e.g. an environment must be inserted before it is populated).

EnvMake automatically infers the necessary dependencies from the thesaurus and initiates (re)compilation and (re-)execution. Hence, there is no notion of an *(Env)Makefile* which has to be created and maintained manually.

8.2 Future Work – Further Utilisation of Thesauri

The idea of a central repository as a vehicle for tool integration is currently being pursued by several software vendors. IBM's AD/Cycle [IBM 1991] is a collection of application development tools and a platform providing services for the integration of these tools. The Repository Manager [IBM 1990] is part of the AD/Cycle framework and provides an interface to a repository containing information utilised by the other tools. DEC, ICL and other companies have similar proposals.

This thesis has demonstrated that the fine-grained, name-based thesauri successfully serve as information repositories for several tools. The provision of persistence enables the thesauri, as well as the tools, to be contained and integrated in the persistent store like any other values. It should be emphasised that because the thesauri are in the same store, the thesaurus can be automatically constructed and updated with guarantees of consistency with the data that they describe.

At present, the *whole* thesaurus is updated regularly or on user request. An *incremental* update can be requested through an interactive interface. Updating the whole thesaurus is inefficient since generally only a fraction of the associated PAS has changed. This problem will be exacerbated as the PASs become larger. The current feature for incremental update does not ensure up-to-date information as it relies on the programmer remembering to analyse the changed programs. When improved hardware is available (Section 8.3), the performance cost of updating the thesaurus during compilation may be affordable. Integrating the thesaurus with the compiler should enable incremental update and ensure up-to-date information.

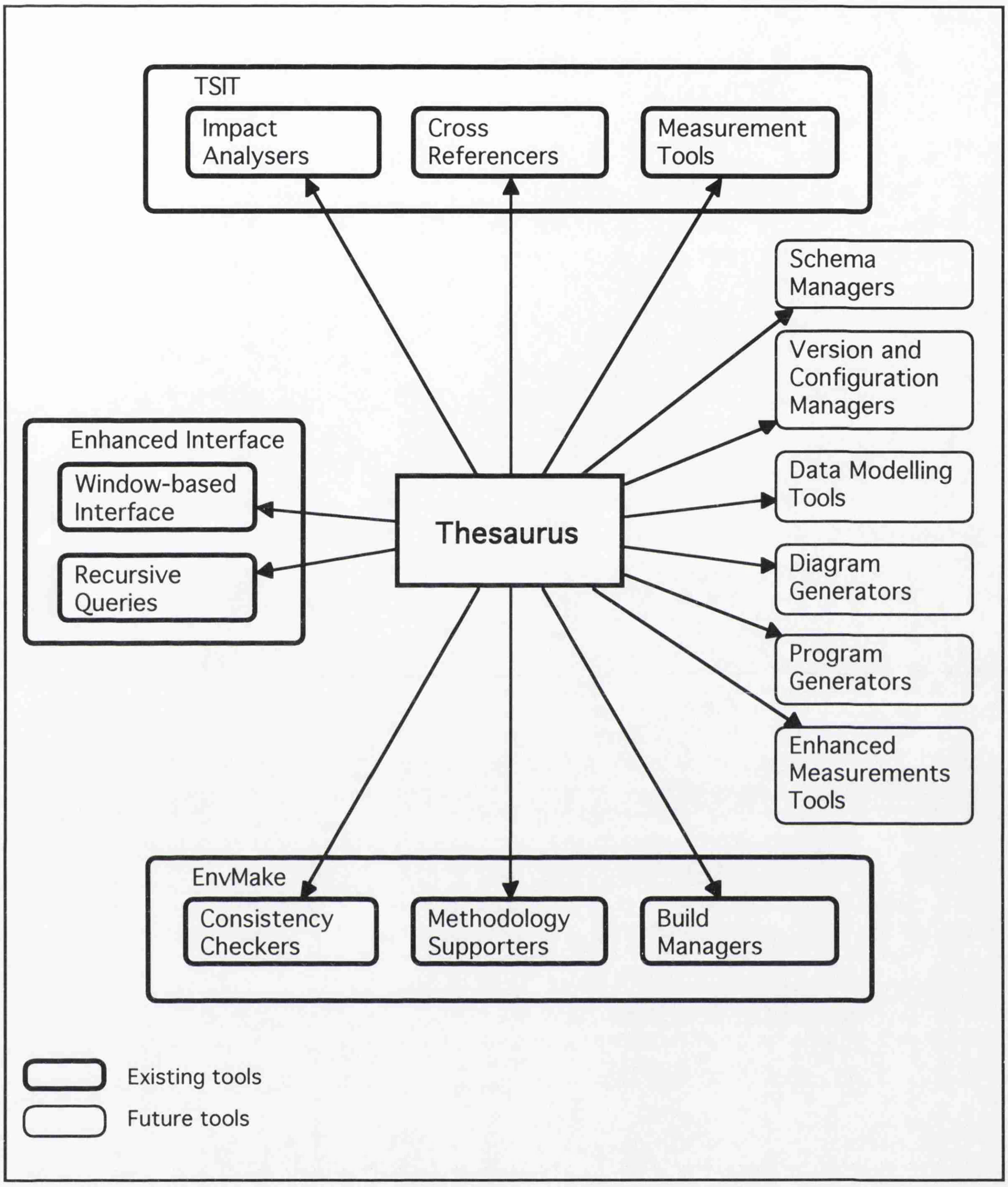


Figure 8.2: More thesaurus-based tools

Several programming support tools, such as EnvMake, have already been built on top of the thesaurus kernel, and various others are expected to follow (Figure 8.2). The work described is thus a step towards a Persistent Software Engineering Environment.

Proposals for future work on schema management (schema evolution), configuration management, some sort of automatic program generation and further measurements are described in the following sections. Figure 8.2 also shows other examples of tools that could benefit from the thesaurus information:

- *Version management:* A thesaurus reflects the state of an application in a certain state. If an application has several versions of its software, then there should be one thesaurus for each version.
- *Data modelling:* A thesaurus could also store information about names related to data modelling [Cooper 1990b, Cooper and Qin 1992]. It could record dependencies between concepts used at the data modelling level and the corresponding implementation at the (low level) programming language level. Ultimately, the thesaurus tool could collect and correlate information from all phases of the software life cycle. It is still crucial, however, that all the information is automatically generated (the tool could operate on design structures, for example) and that the generation is decoupled from the use of other tools.
- *Diagram generation:* The structure of the application programs and persistent store could be visualised in terms of Entity-Relationship diagrams or diagrams of other kinds generated automatically from the thesaurus information.

8.2.1 Schema Evolution

The problem of schema evolution, now identified as a major research issue, arises in any system capable of supporting PASs and is independent of the supported data model. However, Napier88 has a sophisticated type system (as opposed to relational systems, for example) making it a suitable language in which to experiment with strategies for planning and implementing incremental schema change.

> Types provide a way of controlling evolution, by partially verifying programs at each stage. Since typechecking is mechanical, one can guarantee, for a well designed language, that certain classes of errors cannot arise during execution, hence giving a minimal degree of confidence after change. This elimination of entire classes of errors is also very helpful in identifying those problems which cannot be detected during typechecking. [Cardelli 1989a]

Some typing schemes to accommodate schema change in Napier88 have already been proposed [Atkinson 1993]. The challenge is to ensure that *all* consequential changes are dealt with by propagation throughout the system and that no unnecessary changes occur perturbing working practices and operational software. For example, if a new information carrying capacity is added to the schema, programs that do not use it should not change. However, at least one program must be created or changed to collect the data, and all programs that display closely related data should be considered for amendment to show the new data. This will in turn propagate to new screen designs and changed working practices. The semantic difficulties concerning addition require human intervention. It is thus impossible to completely automate the consequences of addition which is the most common kind of change (followed by deletion) according to the HMS measurements. Renaming does not occur so frequently and may be absorbed by organising the software appropriately (though a model for automatic renaming should be relatively simple). In contrast, a model for automatic deletion is conceivable.

TSIT is already an advisory system that returns a list of potential places where the change should be propagated. A more comprehensive analysis including cost estimation of various schema changes is proposed.

Napier88 has structural type equivalence which makes it hard to find all instances of a certain type compared with a language with name equivalence. In languages with name equivalence the type of a value is associated with a certain type declaration, whereas in structural equivalence the type is represented as a graph independent of any type declaration. By using the thesaurus information, one can easily find the definition of a type identifier and all the declarations in which it is being used. However, in any declaration the type can be applied anonymously without using a name for it (although this may be awkward for the programmer if the type is complex).

In the recent Napier88-in-Napier88 compiler [Cutts 1993a] type graphs are stored in environments in the persistent store. Hence, to overcome the problem of structural type equivalence, one possibility would be to let the thesaurus store unique hyper-references[1] (returned by the compiler's type checker) to the type graphs in the persistent store. Extending the thesaurus to contain type information as well would enable various forms of type comparison. Automatic detection of type change might then be possible.

Although changing a procedure type is not a schema change in Napier88 (as opposed to object-oriented systems with procedures or methods defined within a class definition), it is a form of change with potentially serious consequences. For example, all programs calling the actual procedure must be edited. One question is to what extent it is possible to automate or support correct change propagation to all dependent programs.

In addition to tools, also languages should be designed to support evolution. Language design should be influenced by the need to recognise dependencies. As an illustration, consider projection of variants. Programmers may want an alternative to the **project** statement that terminates **complete** requiring all the branches to be processed. If a branch has been added to the variant type, but the programs that use the type have not been changed accordingly, the compiler should give an error message.

8.2.2 Persistent Software Configuration Management

Most software configuration management tools operate independently of data dictionary tools [Holloway 1988b]. One of the problems is that a data dictionary holds fine granularity information, whereas common software configuration management tools operate at the coarse level of files. In a persistent programming environment, code resides in the persistent store in the form of procedures. A candidate for grain size could

1 See hyper-programming, Section 8.2.6.

thus be the procedure. There is, however, no intrinsic difference between procedures and values of other kinds in a persistent environment. An alternative level could therefore be the level of the values that are dealt with in the persistent location binding methodology – typically procedures and complex data structures (a table of geographical information, a list of images, etc.). The challenge is to find a level that gives extensive support, but at the same time is intellectually manageable and does not lead to excess overhead and poor performance [Feldman 1991].

8.2.3 Extensibility of SPASM

The default SPASM constraints, which are checked by EnvMake, may not be adequate or sufficient in all cases. Hence, an enhancement of EnvMake would be to facilitate additional user-defined constraints. One approach to providing such an extensibility would be to allow the user to specify constraints in some kind of formal language. Automatic generation of corresponding Napier88 code for checking the constraints, however, may prove difficult. Alternatively, EnvMake could include a toolkit that supports users in creating the constraint checking code themselves. Extensions in this direction might benefit from work by Stemple and Sheard [Stemple 1989, Sheard and Stemple 1989, Sheard 1991]. Future work should also take into consideration experiences with languages and tools supporting constraint specification such as CCEL [Meyers *et al.* 1993] for C++ and AdaPIC [Wolf *et al.* 1989] and PLEIADES [Tarr and Clarke 1993] developed in the context of Ada.

8.2.4 Automatic Generation of Use-Clauses

Declaring persistent bindings in the scope of a program (use-clauses) is the dominant operation pertinent to environments. This is a tedious task that may impair programming efficiency – particularly for large applications with complex type expressions and deeply nested environments. Recent measurements indicate that use-clauses occupy around 13% of all code (Section 5.7.9). Furthermore, from experience the use-clauses of a new program are often created by copying use-clauses from other programs. This may result in many unused bindings (Section 6.3.3) and thus confusing, verbose and inefficient programs. A use-clause represents a view of an environment (a partial specification of the environment's contents), but the precision in the view identification is lost if the view contains unused bindings as well. Hence, programmers may benefit from tools that support the process of specifying use-clauses.

By utilising the thesaurus information, EnvMake could be enhanced to become such a tool. The design could be as follows.[1] If a potential binding name is passed to Env-

[1] At present, the thesaurus does not provide sufficiently detailed type information for constructed types but could do in the near future.

Make, EnvMake notices whether the name appears in a standard library, local library or elsewhere in the PAS. EnvMake starts searching for a binding with that name in the root environment and then searches downwards, or a search path could be specified by the programmer. If a binding with a matching name is found, EnvMake requests the programmer for acceptance or rejection. If acceptance, EnvMake generates the environment path, the name, constancy and type of the binding. If rejection, the search continues until the correct one is found.[1] If there is no matching occurrence (i.e., the binding to be used is not yet in the persistent store) a warning is given, and the programmer has to complete the use-clause.

There are basically two approaches to how and when EnvMake could generate use-clauses. One approach could be interactive in that every time a programmer needs to declare a binding into the scope of a program, he or she invokes EnvMake with a binding name as parameter and requests a corresponding use-clause template.

Another, probably more convenient, approach would be that the programmer first writes the program without the use-clauses and then requests EnvMake to scan the code and, if possible, to generate the necessary use-clauses for all used identifiers that do not have a corresponding declaration within the program.

As part of program evolution, identifiers denoting persistent bindings may be added to and removed from a program. Hence, if the program has changed, EnvMake should re-generate all the needed use-clauses. In order to simplify the implementation of such a feature, the use-clauses may be constrained to occur all together in the beginning of the program (which complies with the convention already adhered to by most Napier88 programmers).

In a hyper-programming context, the same extension of EnvMake could replace the unsatisfied references by hyper-references directly to the library routines. Given the technology being developed by Munro [Munro 1993], this self-same extension of EnvMake could copy missing library functions from a definitive library store into the intended persistent store. Both of these extensions automate a tedious programming chore and make use of the store and program construction more efficient.

8.2.5 Referencing Environments

The current thesauri identify environments by storing their names. This approach may lead to problems when environments are returned by procedures, when they are elements of vectors, fields of structures, branches of variants, parameters of procedures, etc. or when different identifiers denote the same environment (Section 7.5.2). An alternative

[1] Alternatively, EnvMake could present a list with all matching bindings from which the programmer could select the right one or search exhaustively, and if there were one match, use it. Ambiguity could also be resolved by the programmer specifying search order.

way of identifying environments, that may help solve these problems, is to store direct references to them, i.e., values of type *env* (cf. hyper-references described in the next section). All persistent environments of a PAS should be registered in the thesaurus with direct references.[1] Other occurrences of environments (in use-clauses, drop-clauses, insert-declarations, assignments, etc.) could then be compared with the registered environments by simple equality tests. However, several problems of this approach must be addressed by future research, for example:

i) The environments must exist and be accessible at the time of the analysis. There might be cases where programmers want to perform analyses before the environments have been created. Moreover, since Napier88 at the time of writing does not provide distribution, analysing other people's software (Section 5.7) would be difficult.

ii) The performance would be impaired in some cases and improved in others. For example, if the name of an environment occurs in a use-clause, it is faster to store the (textual) name during the source code analysis than to create a reference to the corresponding environment in the persistent store.

iii) A reference from the thesaurus to the environment would prevent it from being garbage collected even if there were no references from the application programs.

iv) How should the identity of an environment be conveyed to a thesaurus user, if not by its name? For example, Table 5.1 shows that the environment name is indicated after the context attribute of an identifier occurrence (e.g. "UseClause: IO"). In a hyper-programming environment, the name "IO" could be replaced with a "button" that could be clicked to access the environment. Presenting extensive information this way may be impractical; printing the information may be infeasible.

8.2.6 Hyper-Programming

The notion of *persistent hyper-programming* has been introduced in the context of Napier88 [Kirby *et al.* 1992, Kirby 1993]. A hyper-program can directly reference the values and variables in the persistent store over which the program will work. The motivation behind hyper-programming is to provide an integrated programming environment that will support the software engineering process in a better way than is the case in conventional (including persistent) programming environments. It is stated that the advantages of hyper-programming over conventional programming include the following [Kirby *et al.* 1992]:

[1] Actually, the current thesaurus definition (Figure 5.1) allows for registering environments with direct references, but this option has not yet been used.

> ... it allows procedure values to be represented at a source code level; it supports a wider range of binding mechanisms; it allows earlier and more sophisticated type checking; it allows more succinct program representations; and it supports abstract views of source programs.

The flexibility and the interactive nature of the gesture-based hyper-programming environment [Farkas *et al.* 1992] may result in a vast number of references or links and thus a persistent store that is intellectually unmanageable. To alleviate this problem and to facilitate other software engineering needs, a *hyper-world* model has been proposed as a means to impose structure on hyper-programs [Kirby 1993]:

> The hyper-world model offers the programmer a loose coupling mechanism to application spaces or hyper-worlds. Each hyper-world contains the program components and data used by an application, and a schema that describes their relationships. Each hyper-programming system will also have to support additional facilities for 'programming in the large', that is, building large applications from smaller components.

From one viewpoint hyper-programming is conceptually simpler than conventional persistent programming since files and directories are no longer needed. (In hyper-programming the source of a program is also contained in the persistent store.) From another viewpoint, however, it may be conceptually more complex to manage the new notion of link, the more flexible binding mechanisms, the possible hyper-world construct, etc. which add to all the existing constructs of conventional Napier88. In any case, to exploit and benefit from the new technology, there is a need for supporting methodologies and tools. The experience reported in this thesis of developing methodologies and tools for conventional persistent programming will be useful in that respect.

Most of the constraints of the SPASM model are directly applicable to hyper-programming: "all type definitions and their components should be used", "bindings inserted into the persistent store should be used in at least one program",[1] etc. Other constraints may be adapted to the new concepts. For example, "a type name should be declared only once within a PAS" could be changed to "a type name should be declared only once within a hyper-world". (If a type definition with the same name and type expression is defined in two hyper-worlds at the same level,[2] then they should be replaced by one type definition in the enclosing hyper-world at the next level up.) Constraints that are particular to hyper-programming or to the hyper-world model should be added (e.g. constraints on the sort of links allowable *within* a hyper-world and on those *between* hyper-worlds).

Methodologies supporting incremental program construction may be simpler in hyper-programming. For example, in contrast to the persistent location binding

1 This should hold unless the bindings are deliberately created for external use such as library components (Section 6.2.3).

2 A hierarchical structure of hyper-worlds is assumed here.

methodology (Section 6.2.1), creating locations with dummy values (stubs) are no longer necessary[1] – a useful value can be created initially in a convenient way.

An example of something that may be conceptually more complicated is the choice between composition-time, compile-time and run-time binding and checking. This extra flexibility may cause confusion for (at least novice) programmers if not accompanied by methodologies or guidelines indicating when to choose the various alternatives.

Conscious naming and naming conventions are essential for the understanding of software. This issue may be challenged in hyper-programming where significant values are "named" by being located and hence do not have a textual name. In the gesture-based programming environment, a button can represent a link and can be named as a sort of comment insignificant for compilation and execution. As happens with most forms of documentation that is not enforced, programmers may tend to ignore or "forget" to write the link names. If such names exist, however, they could be entered into the thesaurus. Tools could insist on names being used. In general, tools like TSIT and EnvMake in conventional Napier88 should be tailored for and benefit from the new hyper-programming technology. In addition to the traditional user names, the thesaurus must be extended to also contain non-textual "names" in the form of "hyper-identities". Analogously to the dependency information provided by current EnvMake, a similar tool should analyse the reference structure in hyper-programming. Build management involving automatic compilation and execution after change is another task.

The suitability of hyper-programming has yet to be demonstrated. Supporting methodologies and tools are crucial for its success. Future research will investigate whether the hyper-programming environment facilitates sophisticated methodologies and tools in a better way than do conventional (persistent) programming environments.

8.2.7 Further Measurements

In order to turn computing science into a more exact science, more measurements should be obtained provided they are relevant. Claimed problems and proposed solutions should be quantified. Identifying what is interesting to measure and carrying out experiments yielding reliable results, however, are a non-trivial task (cf. the difficulties reported in Section 2.3.3). For example, many human properties that are crucial for change management in large-scale application systems (people's efficiency, skill in management, ability to communicate, etc.) are difficult to measure. We are certain, however, that much more than is the case at present could and should be measured in software engineering in particular and in computing science in general. The thesis is a step in that direction, and further work will also reflect this attitude.

[1] There is one exception, however; dummy locations are needed for mutually recursive procedures.

The HMS system was studied in detail and change statistics were collected by regular measurements [Sjøberg 1993]. By collecting measurements from other systems one might be able to identify properties related to change consequences that are independent of application area, data model and implemented system. Which properties remain constant? For example, in the HMS project the times a field was used appeared to be relatively constant; it varied between 5 and 6 times – independent of the application size and the stage of the development. The number of fields per relation, however, increased with the number of relations. Moreover, one could investigate if there is a relatively fixed ratio between schema changes and consequences for the rest of the system.

As a supplement to anecdotal description of user experiences, attempts should be made to quantify the potential benefits of new and enhanced methodologies and tools. This may be achieved by measuring people's software before and after the methodologies have been adhered to and the supporting tools applied.

To conclude, more information about the extent and kind of change would be useful for further research on change management. In addition to collecting change statistics ourselves, for example by recording differences between versions of the thesaurus for various Napier88 applications, we may also start collecting measurements provided by others [Marche 1993]. All results could eventually be compared in a bigger study on the nature of change.

8.3 Finally

The work described in this thesis concerns maintenance of large-scale, data-intensive application systems. A persistent programming environment has been enhanced with models, methodologies and supporting tools. At first sight application development appears more complicated in a persistent programming language context than in a traditional context. The reason is that issues that earlier were dealt with by the operating system or DBMS, and not made explicit, are now dealt with within the programming language itself. Methodologies and tools are needed whatever the programming environment.

Hopefully, further experimentation will suggest a new, higher-level programming language that approaches the level of conceptual modelling and thus would be more understandable to humans. Such a language should be designed with the purpose of supporting change, and it would benefit from generating strongly typed, persistent code such as Napier88 programs. This vision complies with classical programming language development where a programming language at level *n* enhanced with methodologies and tools at the same level *n* eventually may result in a programming language at level $n + 1$ (Figure 8.3).

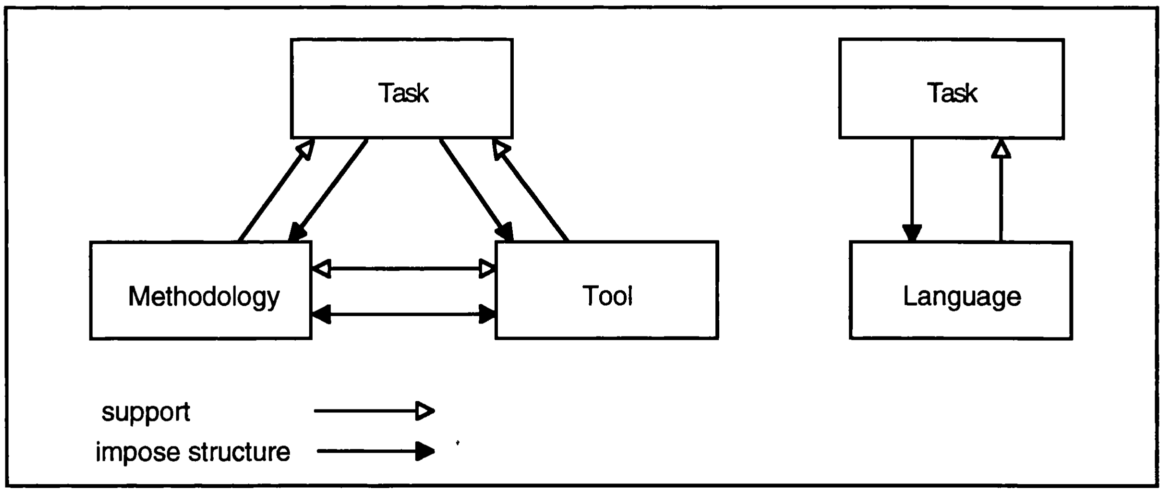


Figure 8.3: Methodologies and tools as input to a new language

Holt [Holt 1993] has identified significant steady growth in the performance of certain components (processes, stores, networks, etc.) supporting computing. For example, his predictions for computers by the end of the decade are given in Table 8.1.

Platform	Feature	Improvement/Year	1993	2001
Desktop	Performance	+50%		50 x
	Memory	+55%	8 MB	256 MB
	Disc	+45%	80 MB	3 GB
Departmental	Performance	+60%		40 x
	Memory	+50%	128 MB	8 GB
	Disc	+50%	30 GB	750 GB
Corporate	Performance	+50%		30 x
	Memory	+50%	10 GB	250 GB
	Disc	+50%	500 GB	15 TB

Table 8.1: Platform improvements

The improvements, however, are not uniform; for example, the speed of transfer between stores improves about one tenth of the increase in store size or processing speed. This may encourage wider use of persistent languages.

A more important mismatch in the rate of improvement lies in our capacity to build and maintain PASs. The improved and cheaper hardware will encourage the production of even larger and more sophisticated PASs. However, our intellectual capacity is not improving at a measurable rate.

The work of this thesis will therefore become ever more relevant. The improved computational power will make use of the proposed tools more economic. The use of tools of this nature for managing change will prove essential as a means of coping with the new scale of systems with current intellectual capacities.

Appendix A: HMS Execution Log

The following excerpt from the execution log shows the number of names of the various name types that were generated on 4 July 1990 for the HMS, BED BUREAU application. The whole generation took about 50 minutes. This was, however, during working hours on a rather loaded machine. Normally, the generation starts at 02:00 and then takes less than 30 minutes.

```
BED BUREAU Wed Jul 04 10:51:42 WETDST 1990:

#####################  Generating from hippo  ###############
cd /usr/hms/test/hippo
cp /users/dag/hms/scripts/hippoGen /usr/hms/test/hippo
hippoGen*
Finding DEFINITIONS of actions, action scripts, and functions, and
USES of action scripts, datums, queries, and update functions ...
Name count from this scan:
    821    821  26516 /users/dag/hms/scripts/hippo.sql
Finding USES of actions (handlers) - the calls ...
Name count after adding the action calls:
    929    929  30238 /users/dag/hms/scripts/hippo.sql
Finding USES of functions - the calls ...
Name count after adding the function calls:
    962    962  31256 /users/dag/hms/scripts/hippo.sql

###################  Second Level, hippo  ##################
Finding DEFINITIONS of actions, action scripts, and functions, and
USES of action scripts, datums, queries, and update functions ...
Name count after adding the first part of the second level:
    962    962  31256 /users/dag/hms/scripts/hippo.sql
Finding USES of actions (handlers) - the calls ...
Name count after adding the action calls, second level:
    962    962  31256 /users/dag/hms/scripts/hippo.sql
Finding USES of functions - the calls ...
Total number of records generated from the hippofiles:
    962    962  31256 /users/dag/hms/scripts/hippo.sql
rm hippoGen*

####################  Generating from screens  ###############
Time:  Wed Jul 04 11:04:08 WETDST 1990
cd /usr/hms/test/screens
cp /users/dag/hms/scripts/screensGen /usr/hms/test/screens
screensGen*
All the macro DEFINITIONS and USES (calls):
     74     74   2331 /users/dag/hms/scripts/screens.sql
Finding the DEFINITIONS of classes and USES of actions, classes, datums, and queries ...
The total number of records generated from the Display Language programs:
   1090   1090  35360 /users/dag/hms/scripts/screens.sql
rm screensGen*
# Getting the DEFINITIONS of the relations and fields from the schema #

SQL*Plus: Version 3.0.6.1.1 - Production on Wed Jul  4 11:27:37 1990
Copyright (c) Oracle Corporation 1979, 1988.  All rights reserved.
Connected to: ORACLE RDBMS V6.0.26.9.1, transaction processing option - Production
SQL> SQL> Disconnected from ORACLE RDBMS V6.0.26.9.1, transaction processing option
- Production
The total number of records and fields:
    333    333  11034 schema.sql

##########  Generating from the Query Dictionary (dd file)..  ########
Time:  Wed Jul 04 11:28:13 WETDST 1990
```

```
The total number of records generated from the Query Dictionary:
   2307   2509  81083 QDThesaurus.sql
The total number of records that will be in the Query_Dictionary relation:
   1026   1026  45020 QueryDictionary.sql
A sorted file of all the generated data for the Thesaurus relation: Insert.sql
The total number of records generated:
   4692   4894 158733 Insert.sql
################ Append to a separate sequential file: ##############
#### Comparisons, creations of deltas and update of the relations.  ###
Time:  Wed Jul 04 11:29:56 WETDST 1990
Unload the THESAURUS relation to the THESAURUS.dat file ...
Connected
********** Executing
select * from THESAURUS
********** Executed

Time:  Wed Jul 04 11:36:00 WETDST 1990
The number of records in the actual Thesaurus relation:
   4551   4751 175689 THESAURUS.dat
 The number of records to be inserted:
    156    158   5267 INSERT.dat
 The number of records to be deleted:
     17    221   2383 DEL.sql
Execute the deletions ...

SQL*Plus: Version 3.0.6.1.1 - Production on Wed Jul  4 11:37:27 1990
Copyright (c) Oracle Corporation 1979, 1988.  All rights reserved.
Connected to: ORACLE RDBMS V6.0.26.9.1, transaction processing option - Production
SQL> SQL>  Disconnected from ORACLE RDBMS V6.0.26.9.1, transaction processing option
- Production
Loading the data into the Thesaurus relation ...
SQL*Loader: Version 1.0.18 - Production on Wed Jul  4 11:38:17 1990
Copyright (c) Oracle Corporation 1979, 1988.  All rights reserved.
Commit point reached - logical record count 44
Commit point reached - logical record count 132
Commit point reached - logical record count 156
The number of records to be inserted into the Versions_Thesaurus relation:
    173    175   7924 VERSIONS_THESAURUS.dat
Loading the data into the Versions_Thesaurus relation ...
SQL*Loader: Version 1.0.18 - Production on Wed Jul  4 11:38:30 1990
Copyright (c) Oracle Corporation 1979, 1988.  All rights reserved.

Commit point reached - logical record count 33
Commit point reached - logical record count 99
Commit point reached - logical record count 165
Commit point reached - logical record count 173
Time:  Wed Jul 04 11:38:56 WETDST 1990
Unload the QUERY_DICTIONARY relation to the QUERY_DICTIONARY.dat file ...
Connected
********** Executing
select * from QUERY_DICTIONARY
********** Executed

Time:  Wed Jul 04 11:39:24 WETDST 1990
The number of records in the actual Query_Dictionary relation:
   1009   1009  44315 DICTIONARY.dat
The number of records to be inserted into Query_Dictionary relation:
     25     25   1185 INSERTQD.dat
The number of records to be deleted from the Query_Dictionary relation:
      8     88   1176 DELQD.sql
Execute the deletions ...

SQL*Plus: Version 3.0.6.1.1 - Production on Wed Jul  4 11:39:32 1990
Copyright (c) Oracle Corporation 1979, 1988.  All rights reserved.
Connected to: ORACLE RDBMS V6.0.26.9.1, transaction processing option - Production
SQL> SQL>  Disconnected from ORACLE RDBMS V6.0.26.9.1, transaction processing option
- Production
Loading the data into the Query_Dictionary relation ...
SQL*Loader: Version 1.0.18 - Production on Wed Jul  4 11:39:59 1990
Copyright (c) Oracle Corporation 1979, 1988.  All rights reserved.
Commit point reached - logical record count 25

Finished: Wed Jul 04 11:40:03 WETDST 1990
```

Appendix B: TSIT Measurements

Kind	Bench -mark	Biblio- graphy	Comp/ TSIT	Eco- Sys	Impl- ADT	Map	Parts- DB	Win	Total
Structure	514 (15.0)	1983 (18.3)	4373 (29.9)	1140 (27.1)	532 (20.1)	1561 (16.5)	644 (37.5)	8361 (22.3)	19108 (22.6)
ProcMono	679 (19.8)	2358 (21.8)	3475 (23.8)	902 (21.4)	96 (3.6)	1012 (10.7)	171 (10.0)	7606 (20.3)	16299 (19.3)
int	469 (13.7)	1674 (15.5)	779 (5.3)	444 (10.5)	278 (10.5)	1004 (10.6)	123 (7.2)	7460 (19.9)	12231 (14.5)
env	411 (12.0)	816 (7.5)	1055 (7.2)	608 (14.4)	147 (5.5)	616 (6.5)	83 (4.8)	6637 (17.7)	10373 (12.3)
Variant	191 (5.6)	669 (6.2)	1259 (8.6)	157 (3.7)	351 (13.2)	540 (5.7)	98 (5.7)	2764 (7.4)	6029 (7.1)
UnboundQuantifier	672 (19.6)	269 (2.5)	858 (5.9)	4 (0.1)	458 (17.3)	2759 (29.1)	78 (4.6)	150 (0.4)	5248 (6.2)
string	120 (3.5)	1305 (12.0)	1575 (10.8)	158 (3.8)	0 (0)	166 (1.8)	30 (1.8)	512 (1.4)	3866 (4.6)
ProcPoly	127 (3.7)	179 (1.7)	208 (1.4)	4 (0.1)	207 (7.8)	964 (10.2)	45 (2.6)	56 (0.2)	1790 (2.1)
image	0 (0)	415 (3.8)	13 (0.1)	102 (2.4)	0 (0)	0 (0)	0 (0)	1104 (2.9)	1634 (1.9)
Vector	45 (1.3)	428 (4.0)	98 (0.7)	226 (5.4)	67 (2.5)	362 (3.8)	14 (0.8)	377 (1)	1617 (1.9)
bool	8 (0.2)	237 (2.2)	283 (1.9)	92 (2.2)	26 (1.0)	176 (1.9)	4 (0.2)	478 (1.3)	1304 (1.5)
any or ?	0 (0)	96 (0.9)	60 (0.4)	32 (0.8)	17 (0.6)	13 (0.1)	6 (0.4)	960 (2.6)	1184 (1.4)
TypeParameter	102 (3.0)	46 (0.4)	139 (1.0)	28 (9.3)	247 (1.0)	102 (1.1)	176 (10.3)	28 (0.7)	868 (1.0)
RecursiveType	61 (1.8)	127 (1.2)	101 (0.7)	87 (2.1)	54 (2.0)	35 (0.4)	164 (9.6)	138 (0.4)	767 (0.9)
null	14 (0.4)	91 (0.8)	203 (1.4)	61 (1.5)	58 (2.2)	32 (0.3)	21 (1.2)	251 (0.7)	731 (0.9)
ParameterisedType	18 (0.5)	12 (0.1)	57 (0.4)	6 (0.1)	114 (4.3)	128 (1.4)	59 (3.4)	23 (0.1)	417 (0.5)
ADT	0 (0)	15 (0.1)	0 (0)	126 (3.0)	0 (0)	0 (0)	0 (0)	234 (0.6)	375 (0.4)
real	0 (0)	0 (0)	45 (0.3)	4 (0.1)	0 (0)	0 (0)	0 (0)	245 (0.7)	294 (0.4)
file	0 (0)	92 (0.9)	31 (0.2)	6 (0.1)	0 (0)	9 (0.1)	0 (0)	110 (0.3)	248 (0.3)
Unb-Witness	0 (0)	26 (0.2)	0 (0)	26 (0.6)	0 (0)	0 (0)	0 (0)	32 (0.1)	84 (0.1)
pixel	0 (0)	0 (0)	9 (0.1)	0 (0)	0 (0)	0 (0)	0 (0)	19 (0.1)	28 (0.0)
pic	0 (0)	0 (0)	5 (0.0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	6 (0.0)
Total	3431 (100.0)	10838 (100.0)	14626 (100.0)	4213 (100.0)	2652 (100.0)	9479 (100.0)	1716 (100.0)	37546 (100.0)	84501 (100.0)

Table B.1: Frequencies of Kind by Application[1]

[1] The table is sorted by the frequencies in the *Total* column. The cells contain number of occurrences and the (column) percentage is given in parentheses.

Context	Bench -mark	Biblio- graphy	Comp/ TSIT	Eco- Sys	Impl- ADT	Map	Parts- DB	WIN	Total
ArgUnaryOpValue	1153	5089	5995	1638	683	3187	450	18218	36413
	(33.6)	(47.0)	(41.0)	(38.9)	(25.8)	(33.6)	(26.2)	(48.5)	(43.1)
TypeNameUse	916	817	2425	383	707	2977	425	2474	11124
	(26.7)	(7.5)	(16.6)	(9.1)	(26.7)	(31.4)	(24.8)	(6.6)	(13.2)
UseClause	380	999	1614	463	94	644	65	3359	7618
	(11.1)	(9.2)	(11.0)	(11.0)	(3.5)	(6.8)	(3.8)	(8.9)	(9.0)
ValueDecl	154	1154	916	336	78	470	83	3535	6726
	(4.5)	(10.6)	(6.3)	(8.0)	(2.9)	(5.0)	(4.8)	(9.4)	(8.0)
StructFieldDeref	79	527	915	252	244	450	141	2730	5338
	(2.3)	(4.9)	(6.3)	(6.0)	(9.2)	(4.7)	(8.2)	(7.3)	(6.3)
ProcParamDecl	100	284	478	70	87	513	33	1346	2911
	(2.9)	(2.6)	(3.3)	(1.7)	(3.3)	(5.4)	(1.9)	(3.6)	(3.4
Assignment	49	658	441	78	43	218	5	1011	2503
	(1.4)	(6.1)	(3.0)	(1.9)	(1.6)	(2.3)	(0.3)	(2.7)	(3.01)
ArgUnaryOpType	25	189	224	424	30	111	35	1036	2074
	(0.7)	(1.7)	(1.5)	(10.1)	(1.1)	(1.2)	(2.0)	(2.8)	(2.5)
StructFieldDecl	67	440	259	196	175	41	174	341	1693
	(2.0)	(4.1)	(1.8)	(4.7)	(6.6)	(0.4)	(10.1)	(0.9)	(2.0)
BindingInserted	70	31	235	31	42	188	15	972	1584
	(2.0)	(0.3)	(1.6)	(0.7)	(1.6)	(2.0)	(0.9)	(2.6)	(1.9)
VariantProjectDyn	8	136	174	16	126	37	36	423	956
	(0.2)	(1.3)	(1.2)	(0.4)	(4.8)	(0.4)	(2.1)	(1.1)	(1.1
ProcQuantifierUse	206	110	260	2	49	237	18	22	904
	(6.0)	(1.0)	(1.8)	(0.0)	(1.8)	(2.5)	(1.0)	(0.1)	(1.1)
ContainsCheck		30	14	31		6		758	839
		(0.3)	(0.1)	(0.7)		(0.1)		(2.0)	(1.0)
TypeDecl	111	103	144	43	60	108	57	58	684
	(3.2)	(1.0)	(1.0)	(1.0)	(2.3)	(1.1)	(3.3)	(0.2)	(0.9)
VariantInject	8	62	137	91	16	65	2	251	632
	(0.2)	(0.6)	(0.9)	(2.2)	(0.6)	(0.7)	(0.1)	(0.7)	(0.7)
BindingDropped		30	11	31	17	7		375	471
		(0.3)	(0.1)	(0.7)	(0.6)	(0.1)		(1.0)	(0.6)
VariantTagRead	4	46	80	5	19	34	10	268	466
	(0.1)	(0.4)	(0.5)	(0.1)	(0.7)	(0.4)	(0.6)	(0.7)	(0.6)
PrimFunctionCall	26	34	83	20	12	26	6	151	358
	(0.8)	(0.3)	(0.6)	(0.5)	(0.5)	(0.3)	(0.3)	(0.4)	(0.4)
VariantTagDecl	23	51	70	21	62	17	48	33	325
	(0.7)	(0.5)	(0.5)	(0.5)	(2.3)	(0.2)	(2.8)	(0.1)	(0.4)
PameterInTypeDecl	32	11	42	7	84	32	59	7	274
	(0.9)	(0.1)	(0.3)	(0.2)	(3.2)	(0.3)	(3.4)	(0.0)	(0.3)
RecursiveTypeDecl	16	19	24	7	24	8	52	15	165
	(0.5)	(0.2)	(0.2)	(0.2)	(0.9)	(0.1)	(3.0)	(0.0)	(0.2)
VariantProjectStatic		1	40	1		77		37	156
		(0.0)	(0.3)	(0.0)		(0.8)		(0.1)	(0.2)
ADTFieldDeref				45				55	100
				(1.1)				(0.1)	(0.1
VariantAlias		2	27	2		26		29	86
		(0.0)	(0.2)	(0.0)		(0.3)		(0.1)	(0.1)
RecursiveValueDecl	4	12	18	1			2	25	62
	(0.1)	(0.1)	(0.1)	(0.0)			(0.1)	(0.1)	(0.1)
ADTalias				17				15	32
				(0.4)				(0.0)	(0.0)
Witness		3		2				2	7
		(0.0)		(0.0)				(0.0)	(0.0)
Total	3431	10838	14626	4213	2652	9479	1716	37546	84501
	(100.0)	(100.0)	(100.0)	(100.0)	(100.0)	(100.0)	(100.0)	(100.0)	(100.0)

Table B.2: Frequencies of Context by Application

#Used	Freq	%	Cum	Cum%
1	38	22.9	38	22.9
2	26	15.6	64	38.5
3	19	11.4	83	50.0
4	11	6.6	94	56.6
5	9	5.4	103	62.0
6	14	8.4	117	70.5
7	1	0.6	118	71.1
8	5	3.0	123	74.1
9	6	3.6	129	77.7
11	3	1.8	132	79.5
12	1	0.6	133	80.1
13	2	1.2	135	81.3
14	2	1.2	137	82.5
16	1	0.6	138	83.1
17	3	1.8	141	84.9
19	1	0.6	142	85.5
20	1	0.6	143	86.1
22	1	0.6	144	86.7
23	3	1.8	147	88.6
24	1	0.6	148	89.2
26	1	0.6	149	89.8
27	1	0.6	150	90.4
34	1	0.6	151	91.0
35	1	0.6	152	91.6
37	1	0.6	153	92.2
38	1	0.6	154	92.8
40	1	0.6	155	93.4
47	1	0.6	156	94.0
51	2	1.2	158	95.2
52	1	0.6	159	95.8
64	1	0.6	160	96.4
73	1	0.6	161	97.0
93	1	0.6	162	97.6
108	1	0.6	163	98.2
130	1	0.6	164	98.8
166	1	0.6	165	99.4
261	1	0.6	166	100.0

Table B.3: Use of type definitions in value instantiations

The *#Used* column indicates the number of times a type definition is used. The *Freq* column contains the number of different type definitions that are used the number of times indicated by the *#Used* column. For example, the second row shows that there are 26 type definitions (15.6% of all type definitions) that are used exactly twice.

Context	Application	Programs	Min	Max	Mean	Std	Sum
BindingInserted	Benchmark	11	1	3	1.5	0.7	16
	Bibliography	26	1	1	1.0	.	26
	Comp/TSIT	11	1	4	1.7	0.9	19
	EcoSys	20	1	1	1.0	.	20
	ImplADT	8	1	3	1.5	0.9	12
	Map	11	1	4	1.5	0.9	16
	PartsDB	2	1	5	3.0	2.8	6
	WIN	148	1	13	2.7	2.3	406
	Total	237	1	13	2.2	1.8	521
BindingDropped	Bibliography	25	1	1	1.0	.	25
	Comp/TSIT	9	1	2	1.1	0.3	10
	EcoSys	20	1	1	1.0	.	20
	ImplADT	3	1	1	1.0	.	3
	Map	3	1	3	2.0	1.0	6
	WIN	147	1	3	2.5	1.2	371
	Total	207	1	3	2.1	0.9	435
UseClause	Benchmark	26	3	8	6.2	1.4	162
	Bibliography	33	1	19	10.2	4.3	336
	Comp/TSIT	76	2	12	7.2	2.4	544
	EcoSys	20	4	20	11.5	6.7	230
	ImplADT	11	2	5	3.4	1.0	37
	Map	24	1	11	7.3	3.0	174
	PartsDB	4	3	9	6.0	2.4	24
	WIN	151	4	23	9.7	4.7	1468
	Total	345	1	23	8.6	3.9	2975
ContainsCheck	Bibliography	25	1	1	1.0	.	25
	Comp/TSIT	10	1	2	1.1	0.3	11
	EcoSys	20	1	1	1.0	.	20
	Map	3	1	2	1.7	0.6	5
	WIN	148	1	3	2.5	1.2	375
	Total	206	1	3	2.1	0.9	436

Table B.4: Environments accessed per program[1]

The *Programs* column contains the number of programs involving identifiers occurring in the respective contexts. *Sum* is the number of unique (program name, environment name) pairs.

[1] Applications without identifiers in the respective contexts are omitted from the table.

Context	Application	Envs	Min	Max	Mean	Std	Sum
BindingInserted	Benchmark	5	1	6	3.2	1.9	16
	Bibliography	1	26	26	26.0	.	26
	Comp/TSIT	9	1	5	2.1	1.7	19
	EcoSys	4	2	11	5.0	1.0	20
	ImplADT	6	1	3	2.0	0.9	12
	Map	11	1	3	1.5	0.8	16
	PartsDB	5	1	2	1.2	0.4	6
	WIN	40	1	143	10.2	12.0	406
	Total	81	1	143	6.4	8.5	521
BindingDropped	Bibliography	1	25	25	25.0	.	25
	Comp/TSIT	4	1	4	2.5	1.7	10
	EcoSys	4	2	11	5.0	1.0	20
	ImplADT	1	3	3	3.0	.	3
	Map	3	2	2	2.0	.	6
	WIN	16	1	142	23.2	16.7	371
	Total	29	1	142	15.0	12.5	435
UseClause	Benchmark	14	1	26	11.6	6.5	162
	Bibliography	26	1	33	12.9	10.2	336
	Comp/TSIT	30	1	75	18.1	15.2	544
	EcoSys	32	1	20	7.2	3.5	230
	ImplADT	9	1	6	4.1	1.9	37
	Map	18	1	24	9.7	6.2	174
	PartsDB	10	1	4	2.4	1.1	24
	WIN	49	1	151	30.0	31.2	1468
	Total	188	1	151	15.8	17.7	2975
ContainsCheck	Bibliography	1	25	25	25.0	.	25
	Comp/TSIT	5	1	4	2.2	1.6	11
	EcoSys	4	4	11	5.0	1.0	20
	Map	4	1	2	1.3	0.5	5
	WIN	18	1	142	20.8	16.2	375
	Total	32	1	142	13.6	12.1	436

Table B.5: Programs per environment

Bibliography

[Acheampong 1993] Acheampong, I., Persistent Programming Language Support for Information (Bibliographic) Retrieval., MSc thesis in preparation, Computing Science Department, University of Glasgow, 1993.

[Adams *et al.* 1989] Adams, R., Weinert, A. and Tichy, W., "Software Change Dynamics *or* Half of all Ada Compilations are Redundant", *European Software Engineering Conference*, 1989.

[Agresti and Evanco 1992] Agresti, W.W. and Evanco, W.M., "Projecting Software Defects from Analyzing Ada Designs", *IEEE Transactions on Software Engineering*, Vol. SE-18, No. 11, pp. 988–997, November 1992.

[Ahlsen *et al.* 1983] Ahlsén, M., Björnerstedt, A., Britts, S., Hultén, C. and Söderlund, L., "Making Type Changes Transparent", *Proceedings of IEEE Workshop on Languages for Automation, Chicago*, pp. 110–117, IEEE Computer Society Press, November 1983.

[Albano 1983] Albano, A., "Type Hierarchies and Semantic Data Models", *ACM SIGPLAN Notices*, Vol. 18, No. 6, pp. 178–186, 1983.

[Albano *et al.* 1985] Albano, A., Cardelli, L. and Orsini, R., "Galileo: A Strongly Typed, Interactive Conceptual Language", *ACM Transactions on Database Systems,* Vol. 10, No. 2, pp. 230–260, June 1985.

[Allen *et al.* 1982] Allen, F.W., Loomis, M.E.S. and Mannino, M.V., "The Integrated Dictionary/Directory System", *ACM Computing Surveys*, Vol. 14, No. 2, pp. 245–286, June 1982.

[ANSI 1988] ANSI X3.138-1988 Information Resource Dictionary System (IRDS), October 1988.

[Archer and Devlin 1986] Archer, J.E. and Devlin, M.T., "Rational's Experience using Ada for Very Large Systems", *Proceedings First International Conference on Ada Applications for the NASA Space Station*, 1986.

[Ariav 1991] Ariav, G., "Temporally Oriented Data Definitions: Managing Schema Evolution in Temporally Oriented Databases", *Data and Knowledge Engineering,* Vol. 6, No. 6, pp. 451–467, October 1991.

[Atkinson 1978] Atkinson, M.P., "Programming Languages and Databases", *Proceedings Fourth International Conference on Very Large Data Bases (Berlin, West Germany, 13th–15th September 1978),* S.B. Yao (editor), pp. 408–419, IEEE and ACM, 1978.

[Atkinson 1989] Atkinson, M.P., "Questioning Persistent Types", *Proceedings of Second International Workshop on Database Programming Languages (Salishan Lodge, Oregon, June 1989),* Hull, R., Morrison, M. and Stemple, D. (editors), pp. 2–24, Morgan Kaufmann Publishers, San Mateo, CA, 1989.

[Atkinson 1990] Atkinson, M.P., "The Principles and Problems of Database Research", *Proceedings of the 1990 Glasgow Database Workshop*, Cooper, R., Stewart, A. and Trinder, P. (editors), pp. 1–12, Technical Report CSC 90/R16, Computing Science Department, University of Glasgow, March 1990.

[Atkinson 1992] Atkinson, M.P., "Persistent Foundations for Scalable Multi-Paradigmal Systems", Invited paper, *Distributed Object Management (Edmonton, Alberta,* *Canada, 18th–21st August 1992)*, Özsu, M.T., Dayal, U., and Valduriez, P. (editors), Morgan Kaufmann, 1992.

[Atkinson 1993] Atkinson, M.P., Lecture Notes in Napier88 Programming, Computing Science Department, University of Glasgow, 1993.

[Atkinson and Buneman 1987] Atkinson, M.P. and Buneman, O.P., "Types and Persistence in Database Programming Languages", *ACM Computing Surveys*, Vol. 19, No. 2, pp. 105–190, 1987.

[Atkinson and Morrison 1985] Atkinson, M.P. and Morrison, R., "Procedures as Persistent Data Objects", *ACM Transactions on Programming Languages and Systems,* Vol. 7, No. 4, pp. 539–559, 1985.

[Atkinson and Morrison 1986] Atkinson, M.P. and Morrison, R., "Integrated Persistent Programming Systems", *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pp. 842–854, January 1986.

[Atkinson *et al.* 1982] Atkinson, M.P., Chisholm, K.J. and Cockshott, W.P., "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, Vol. 17, No. 7, pp. 24–31, July 1982.

[Atkinson *et al.* 1983a] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R., "An Approach to Persistent Programming", *The Computer Journal,* Vol. 26, No. 4, pp. 360–365, November 1983.

[Atkinson *et al.* 1983b] Atkinson, M.P., Chisholm, K.J. and Cockshott, W.P., "CMS – A Chunk Management System", *Software – Practice and Experience*, Vol. 13, No. 3, pp. 273–285, March 1983.

[Atkinson *et al.* 1983c] Atkinson, M.P., Chisholm, K.J., Cockshott, W.P. and Marshall, R.M., "Algorithms for a Persistent Heap", *Software – Practice and Experience*, Vol. 13, No. 3, pp. 259–272, March 1983.

[Atkinson *et al.* 1988] Atkinson, M.P., Buneman, O.P. and Morrison, R., "Binding and Type Checking in Database Programming Languages", *The Computer Journal*, Vol. 31, No. 2, pp. 99–109, 1988.

[Atkinson *et al.* 1990] Atkinson, M.P., Richard, P. and Trinder, P.W., "Bulk Types for Large Scale Programming", In *Next Generation Information System Technology: Proceedings of the First International East/West Database Workshop (Kiev, USSR, 9th–12th October 1990)*, Schmidt, J.W. and Stogny, A.A. (editors), pp. 228–250, Lecture Notes in Computer Science 504, Springer-Verlag, 1991.

[Atkinson *et al.* 1991a] Atkinson, M.P., Lecluse, C., Philbrow, P. and Richard, P., "Design Issues in a Map Language", *Proceedings of the Third International Workshop on Database Programming Language (Nafplion, Greece, 27th–30th August 1991)*, Kanellakis, P. and Schmidt, J.W. (editors), pp. 20–32, Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Atkinson *et al.* 1991b] Atkinson, M.P., Lecluse, C., Philbrow, P.C., and Richard, P., "Maps as Bulk Types for Data Base Programming Languages", *Proceedings of the Annual Esprit Conference*, pp. 731–757, 1991.

[Atkinson *et al.* 1993] Atkinson, M.P., Bailey, P.J., Jackson, N. and Philbrow, P.C., Napier88 Libraries, Technical report in preparation, ESPRIT Basic Research Action, Project Number 6309 – FIDE, 1993.

[Bachman 1988] Bachman, C., "A CASE for Reverse Engineering", *Datamation*, Vol. 34, No. 13, pp. 49–56, July 1988.

[Bailey 1989] Bailey, P.J., "Performance Evaluation in a Persistent Object System", In *Persistent Object Stores (Proceedings of the Third International Workshop, 10th–*

13th January 1989, Newcastle, New South Wales, Australia), Rosenberg, J. and Koch, D. (editors), pp. 289–299, Springer-Verlag and British Computer Society, 1989.

[Banerjee *et al.* 1987] Banerjee, J., Kim, W., Kim, H.-J. and Korth, H.F., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", *Proceedings of the ACM SIGMOD 1987 Conference on the Management of Data (San Francisco, CA, 27th–29th May 1987)*, pp. 311–322, 1987.

[Barclay *et al.* 1992] Barclay, P.J., Fraser, C.M. and Kennedy, J.B, "Using a Persistent System to Construct a Customised Interface to a Ecological Database", *International Workshop on Interfaces to Databases (Glasgow, 1st–3rd, July 1992),* Cooper, R.L. (editor), pp. 225–243, Workshops in Computer Science, Springer-Verlag, June 1993.

[Barnard *et al.* 1982] Barnard, P., Hammond, N.V., MacLean, A. and Morton, J., "Learning and Remembering Interactive Commands", *Proceedings of Conference on Human Factors in Computer Systems*, ACM Washington, CD, 1982.

[Batini *et al.* 1986] Batini, C., Lenzerini, M and Navathe, S.B., "A Comparative Analysis of Methodologies for Database Schema Integration", *ACM Computing Surveys*, Vol. 18, No. 4, pp. 323–364, April 1986.

[Baxter 1992] Baxter, I.D., "Design Maintenance Systems", *Communications of the ACM,* Vol. 35, No. 4, pp. 73–89, April 1992.

[Berman 1991] Berman, S., P-Pascal: A Data-Oriented Persistent Programming Language, Department of Computer Science, University of Cape Town, August 1991.

[Birnie 1991] Birnie, A., Sun Engineering Database Benchmark, 2nd Annual FIDE Review Meeting, Computing Science Department, University of Glasgow, September, 1991.

[Bjørner 1991] Bjørner, D., "Formal Methods in Software Development – Requirements for a CASE", In *Software Development Environments and CASE Technology, European Symposium (Germany, June 1991)*, Endres, A, and Weber, H. (editors), pp. 178–210, Lecture Notes in Computer Science 509, Springer-Verlag, 1991.

[Boehm 1988] Boehm, B.W., "A Spiral Model of Software Development and Enhancement", *IEEE Computer,* Vol. 21, No. 5, May 1988.

[Bott 1989] Bott, F. (editor), *ECLIPSE: An Integrated Project Support Environment,* IEE Computing Series 14, Peter Peregrinus, 1989.

[Bourne 1979] Bourne, T.J., "The Data Dictionary System in Analysis and Design", *ICL Technical Journal*, Vol. 1, No. 3, pp. 292–298, November 1979.

[Bratsberg 1993] Bratsberg, S.E., Evolution and Integration of Classes in Object-Oriented Databases, PhD thesis, The Norwegian Institute of Technology, University of Trondheim, Norway, June 1993.

[Brodie 1992] Brodie, M., "The Promise of Distributed Computing and the Challenges of Legacy Systems", Invited paper, *Tenth British National Conference on Databases (Aberdeen, Scotland, 6th–8th July),* Gray, P.M.D. and Lucas, R.J. (editors), pp. 1–28, Lecture Notes in Computer Science 618, Springer-Verlag, 1992.

[Brooks 1975] Brooks, F.P., *The Mythical Man-Month*, Addison Wesley, 1975.

[Brown 1989] Brown, A.L., Persistent Object Stores, PhD thesis, Department of Mathematical and Computational Sciences, University of St Andrews, 1989.

[Brunhoff 1991] Brunhoff, T., Makedepend Manual Page, Tektronix, Inc. and MIT Project Athena, University of New Mexico, April 1991.

[Buxton 1980] Buxton, J.N., Requirements for Ada Programming Support Environments – "Stoneman", Technical Report, US Department of Defence, Washington DC, 1980.

[Cardelli 1989a] Cardelli, L., Typeful Programming, Digital Systems Research Center Report 45, Digital Equipment Corporation, Systems Research Centre, Palo Alto, CA, USA, May 1989.

[Cardelli 1989b] Cardelli, L., The Quest Language and System (Tracking Draft), Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, USA, August 1989.

[Cardelli and Wegner 1985] Cardelli, L. and Wegner, P., "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys,* Vol. 17, No. 4, pp. 471–522, December 1985.

[Cartmell and Alderson 1989] Cartmell, J. and Alderson, A., "The Eclipse Two-Tier Database", In *ECLIPSE: An Integrated Project Support Environment*, Bott, E. (editor), pp. 39–67, IEE Computing Series 14, 1989.

[Casais 1991] Casais, E., Managing Evolution in Object Oriented Environments: An Algorithmic Approach, PhD thesis, Faculté des sciences économiques et sociales, University of Geneva, 1991.

[Chapin 1988] Chapin, N., "Software Maintenance Life Cycle", *Proceedings Conference on Software Maintenance (Phoenix, AR, USA, 24th–27th October 1988)*, pp. 6–13, IEEE Computer Society Press, 1988.

[Chen 1976] Chen, P.P., "The Entity-Relationship Model – Toward a Unified View of Data", *ACM Transactions on Database Systems*, Vol. 1, No. 1, pp. 9–36, 1976.

[Chikofsky and Cross 1990] Chikofsky and Cross, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, January 1990.

[Clifton 1990] Clifton, N., Display Language Documentation, October 1990.

[Colbrook and Smythe 1989] Colbrook, A. and Smythe, C., "The Retrospective Introduction of Abstraction in Software", *Proceedings of Conference on Software Maintenance (Miami, FL, USA, 16th–19th October 1989)*, pp. 166–173, IEEE Computer Society Press, Los Alamitos, CA, 1989.

[Collofello and Buck 1987] Collofello, J.S. and Buck, J.J., "Software Quality Assurance for Maintenance", *IEEE Software*, pp. 46–51, September 1987.

[Connor 1991] Connor, R.C.H, Types and Polymorphism in Persistent Programming Systems, PhD thesis, Department of Mathematical and Computational Sciences, University of St Andrews, 1991.

[Connor *et al.* 1990] Connor, R.C.H., Brown, A.L., Cutts, Q.I., Dearle, A., Morrison, R. and Rosenberg, J., "Type Equivalence Checking in Persistent Object Systems", *Proceedings of the Fourth International Workshop on Persistent Object Systems, Their Design, Implementation and Use (Martha's Vineyard, USA, September 1990),* Dearle, A., Shaw, G.M. and Zdonik, S.B. (editors), pp. 154–167, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[Connor *et al.* 1991] Connor, R.C.H., McNally, D. and Morrison, R., "Subtyping and Assignment in Database Programming Languages", *Proceedings of the Third International Workshop on Database Programming Languages (Nafplion, Greece, 27th–30th August 1991),* Kanellakis, P. and Schmidt, J.W. (editors), pp. 363–382, Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Constantine and Yourdon 1979] Constantine, L.L. and Yourdon, E., *Structured Design,* Englewood Cliffs, N.J. Prentice-Hall, 1979.

[Cooper 1990a] Cooper, R.L., On the Utilisation of Persistent Programming Environments, PhD thesis, Department of Computing Science, University of Glasgow, 1990.

[Cooper 1990b] R.L. Cooper, "Configurable Data Modelling Systems", *Proceedings of the Ninth International Conference on the Entity Relationship Approach* (*Lausanne, Switzerland, 8th–10th October 1990*), pp. 35–52, 1990.

[Cooper and Qin 1992] Cooper, R.L. and Qin, Z., "A Graphical Data Modelling Program with Constraint Specification and Management", *Tenth British National Conference on Databases (Aberdeen, Scotland, 6th–8th July),* Gray, P.M.D. and Lucas, R.J. (editors), pp. 192–208, Lecture Notes in Computer Science 618, Springer-Verlag, 1992.

[Copeland and Maier 1984] Copeland, G. and Maier, D., "Making Smalltalk a Database System", *Proceedings of the ACM SIGMOD 1984 Conference on the Management of Data (Boston, Mass., 18th–21st June), ACM SIGMOD Record,* Vol. 14, No. 2, pp. 316–325, June 1984.

[Cutts 1993a] Cutts, Q.I., Delivering the Benefits of Persistence to System Construction and Execution, PhD thesis, Department of Mathematical and Computational Sciences, University of St Andrews, 1993.

[Cutts 1993b] Cutts, Q.I., Private Communication, 1993.

[Cutts *et al.* 1990] Cutts, Q.I., Dearle, A. and Kirby, G.N.C., WIN Programmers' Manual, Research Report CS/90/17, University of St Andrews, 1990.

[Dahl *et al.* 1972] Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R., *Structured Programming,* A.P.I.C. Studies in Data Processing No. 8, Academic Press, New York, 1972.

[Dart 1991] Dart, S., "Concepts in Configuration Management Systems", *Proceedings Third International Workshop on Software Configuration Management (Trondheim, Norway, 12th–14th June 1991),* pp. 1–18, 1991.

[Dart *et al.* 1987] Dart, S.A., Ellison, R.J., Feiler, P.H. and Habermann, A.N., "Software Development Environments", *IEEE Computer,* Vol. 20, No. 11, pp. 18–28, November 1987.

[Date 1990] Date, C.J., *An Introduction to Database Systems,* Volume 1, Fifth edition, Addison Wesley, 1990.

[Davie and Morrison 1981] Davie, A.J.T. and Morrison, R., *Recursive Descent Compiling,* Ellis Horwood Publishers, 1981.

[DDSWP 1977] Data Dictionary Systems Working Party, Report British Computer Society, March 1977.

[Dearle 1987] Dearle, A., "Constructing Compilers in a Persistent Environment", *Proceedings of the Second International Workshop on Persistent Object Systems: Their Design, Implementation and Use (Appin, Scotland, 25th–28th August 1987),* Research Report PPRR-44-87, Universities of Glasgow and St Andrews, 1987.

[Dearle 1988] Dearle, A., On the Construction of Persistent Programming Environments, PhD thesis, Department of Mathematical and Computational Sciences, University of St Andrews, 1988.

[Dearle *et al.* 1989] Dearle, A., Connor, R., Brown, A.L. and Morrison, R., "Napier88 – A Database Programming Language?", *Proceedings of Second International Workshop on Database Programming Languages (Salishan Lodge, Oregon, June 1989),* Hull, R., Morrison, M. and Stemple, D. (editors), pp. 179–195, 1989.

[Dearle *et al.* 1992] Dearle, A., Cutts, Q. and Connor, R., An Application Architecture using Type-Safe Incremental Linking, Technical Report FIDE/92/56, ESPRIT Basic Research Action, Project Number 6309 – FIDE, 1992.

[DEC 1989] VAX Language-Sensitive Editor and VAX Source Code Analyzer User Manual, AA-PAJLA-TK, Digital Equipment Corporation, 1989.

[DEC 1993] DEC FUSE Handbook, AA-PF4TA-TE, Digital Equipment Corporation, 1993.

[DeMarco 1979] DeMarco, T., *Structured Analysis and System Specification*, Englewood Cliffs, N.J. Prentice-Hall, 1979.

[DeRemer and Kron 1976] DeRemer, F. and Kron, H.H., "Programming-in-the-Large versus Programming-in-the-Small", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2, pp. 80–86, June 1976.

[Dolk and Kirsch 1987] Dolk, D.R. and Kirsch, R.A., "A Relational Information Resource Dictionary System", *Communications of the ACM*, Vol. 30, No. 1, pp. 48–61, January 1987.

[Dolotta *et al.* 1978] Dolotta, T.A., Haight, R.C., Mashev, J.R., "The Programmer's Workbench", *Bell Systems Technical Journal*, Vol. 57, No. 6, pp. 2177–2200, 1978.

[ECMA 1990] European Computer Manufacturers' Association (ECMA), Technical Report ECMA-149, December 1990.

[EIA 1991] CDIF – Framework for Modeling and Extensibility, EIA-PN2387, July 1991.

[Elshoff 1976] Elshoff, J.L., "An Analysis of some Commercial PL/1 Programs", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2, pp. 113–120, June 1976.

[England and Selwyn 1990] England, A. and Selwyn, B., Hippo Language Guide, Perihelion Software Ltd., November 1990.

[Farkas *et al.* 1992] Farkas, A., Dearle, A., Kirby, G.N.C., Cutts, Q.I., Morrison, R. and Connor, R.C.H., Persistent Program Construction through Browsing and User Gesture with some Typing, Technical Report FIDE/92/52, ESPRIT Basic Research Action, Project Number 6309 – FIDE, 1992.

[Fegaras and Stemple 1991] Fegaras, L. and Stemple, D., "Using Type Transformation in Database System Implementation", *Proceedings of the Third International Workshop on Database Programming Language (Nafplion, Greece, 27th–30th August 1991)*, Kanellakis, P. and Schmidt, J.W. (editors), pp. 337–356, Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Fegaras *et al.* 1989] Fegaras, L., Sheard, T. and Stemple, D., "The ADABTPL Type System", *Proceedings of Second International Workshop on Database Programming Languages (Salishan Lodge, Oregon, June 1989)*, Hull, R., Morrison, M. and Stemple, D. (editors), pp. 207–218, 1989.

[Feldman 1979] Feldman, S.I., "Make – A Program for Maintaining Computer Programs", *Software – Practice and Experience*, Vol. 9, No. 4, pp. 255–265, April 1979.

[Feldman 1991] Feldman, S.I., "Software Configuration Management: Past Uses and Future Challenge", *Proceedings of Third European Software Engineering Conference* (*Milan, Italy, October 1991*), Lamsweerde A. van and Fugetta A. (editors), pp. 1–6, Lecture Notes in Computer Science 550, Springer-Verlag, 1991.

[Ferraby 1991] Ferraby, L., *Change Control During Computer Systems Development*, Prentice-Hall (UK), 1991.

[Fosdick and Osterweil 1976] Fosdick, L.D. and Osterweil, L.J., "Data Flow Analysis in Software Reliability", *ACM Computing Surveys*, Vol. 8, No. 3, pp. 305–330, 1976.

[Glφersen 1993] Glφersen, R., Private Communication, Statistics Norway, Oslo, Norway, April 1993.

[Goldberg 1984] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, 1984.

[Gopal *et al.* 1992] Gopal, R., Prasad, R. and Gopal, R., "Supporting System Maintenance with Automatic Decomposition Schemes", *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences,* pp. 507–516, January 1992.

[Greenwood *et al.* 1992] Greenwood, R.M., Guy, M.R. and Robinson, D.J.K., "The Use of a Persistent Language in the Implementation of a Process Support System", *ICL Technical Journal*, Vol. 8, No. 1, pp. 108–130, May 1992.

[Griswold and Notkin 1992] Griswold, W.G. and Notkin, D., "Computer-Aided vs. Manual Program Restructuring", *ACM Software Engineering Notes,* Vol. 17, No. 1, pp. 33–41, January 1992.

[Habermann and Notkin 1986] Habermann, A.V. and Notkin, D., "Gandalf: Software Development Environments", *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 2, pp. 1117–1127, December 1986.

[Holloway 1988a] Holloway, S., *The Future of Data Dictionaries, DATABASE 88 (19th–20th May 1988, Open University, Milton Keynes),* Gower Technical, The British Computer Society Database Specialist Group, 1988.

[Holloway 1988b] Holloway, S., "Reporting from Data Dictionaries", In *The Future of Data Dictionaries, DATABASE 88 (19th–20th May 1988, Open University, Milton Keynes)*, Holloway, S. (editor), pp. 69–92, Gower Technical, The British Computer Society Database Specialist Group, 1988.

[Holt 1993] Holt, N., High Technology Trends, Seminar, The 1993 IT Summit, 22th–24th June, Glasgow, 1993.

[Humphrey 1989] *Managing the Software Process*, SEI Series, Addison-Wesley, 1989.

[IBM 1978] IBM Internal Report on the Contents of a Sample of Programs Surveyed, IBM Research Centre San Jose, California, 1978.

[IBM 1980] DB/DC Data Dictionary General Information Manual, GH20-9104-3, IBM, 1980.

[IBM 1990] Repository Manager/MVS, General Information, GC26-4608-1, IBM, 1990.

[IBM 1991] IBM SAA AD/Cycle Concepts, GC26-4531-01, IBM, 1991.

[IBM 1992] The Information Management Library: Problem, Change, and Configuration Management, User's Guide, SC34-4328-00, IBM, March 1992.

[Imber 1991] Imber, M., "The CASE Data Interchange Format (CDIF) Standards", In *Software Engineering Environments: Vol. 3,* Long, F. (editor), pp. 457–474, Ellis Horwood Limited, Chichester, England, 1991.

[ISO 1990] ISO/IEC 10027: Information Resource Dictionary System (IRDS) Framework, 1990.

[Jackson 1975] Jackson, M.A., *Principles of Program Design,* A.P.I.C. Studies in Data Processing No. 12, Academic Press, London, 1975.

[Jackson 1983] Jackson, M., *System Development*, Englewood Cliffs, N.J. Prentice-Hall, 1983.

[Jacobs and Hull 1991] Jacobs, D. and Hull, R., "Database Programming with Delayed Updates", *Proceedings of the Third International Workshop on Database Programming Language (Nafplion, Greece, 27th–30th August 1991)*, Kanellakis, P. and Schmidt, J.W. (editors), pp. 416–428, Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Jandrasics 1981] Jandrasics, G. "SOFTDOC – A System for Automated Software Analysis and Documentation", *Proceedings ACM Workshop on Software Quality Assurance*, April 1981.

[Kay 1992] Kay, M.H., "The Architecture of an Open Dictionary", *ICL Technical Journal*, Vol. 8, No. 1, pp. 85–107, May 1992.

[Keables *et al.* 1988] Keables, J., Roberson, K. and von Mayrhauser, A., "Data Flow Analysis and its Application to Software Maintenance", *Proceedings Conference on Software Maintenance (Phoenix, AR, USA, 24th–27th October 1988)*, pp. 335–347, IEEE Computer Society Press, 1988.

[Kim and Chou 1988] Kim, W. and Chou, H.T., "Versions of Schema for Object-Oriented Databases", *Proceedings of Fourteenth Conference on Very Large Databases,* Los Angeles, 1988.

[King 1967] King, P.J.H., "Some Comments on Systematics", *The Computer Journal*, Vol. 10, pp. 116–118, 1967.

[King 1969] King, P.J.H., "System Analysis Documentation: Computer-Aided Data Dictionary Definition", *The Computer Journal*, Vol. 12, No. 1, pp. 6–9, 1969.

[Kirby 1993] Kirby, G.N.C., Reflection and Hyper-Programming in Persistent Programming Systems, PhD thesis, Department of Mathematical and Computational Sciences, University of St Andrews, 1993.

[Kirby and Dearle 1990] Kirby, G.N.C. and Dearle, A., An Adaptive Graphical Browser for Napier88, Research Report CS/90/16, Department of Mathematical and Computational Sciences, University of St Andrews, 1990.

[Kirby *et al.* 1992] Kirby, G., Connor, R., Cutts, Q., Dearle, A., Farkas, A. and Morrison, R., "Persistent Hyper-Programs", *Proceedings Fifth International Workshop on Persistent Object Systems. Design, Implementation and Use (San Miniato, Italy, 1st–4th September 1992)*, Albano, A. and Morrison, R. (editors), pp. 86–106, Springer-Verlag in collaboration with the British Computer Society, 1992.

[Knuth 1972] Knuth, D.E., "An Empirical Study of FORTRAN Programs", *Software – Practice and Experience*, Vol. 1, No. 2, pp. 105–133, April–June 1971.

[Knuth 1973] Knuth, D.E., *Fundamental Algorithms*, Vol. 1, In series *The Art of Computer Programming*, Addison-Wesley, January 1973.

[Krueger 1992] Krueger, C.W., "Software Reuse", *ACM Computing Surveys*, Vol. 24, No. 2, pp. 131–183, June 1992.

[Leblang *et al.* 1985] Leblang, D.B., Chase, R.P. and McLean, G.D., "The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts", *First Conference on Computer Workstations,* pp. 266–280, IEEE, November 1985.

[Lehman 1976] Lehman, M.M., "Human Thought and Action as an Ingredient of System Behaviour", In *Encyclopædia of Ignorance*, Duncan, R. and Weston-Smith, M. (editors), pp. 347–354, Pergamon Press, Oxford, 1976. (Reprinted in [Lehman and Belady 1985], pp. 237–246.)

[Lehman 1978] Lehman, M.M., "Laws of Program Evolution – Rules and Tools for Programming Management", *Proceedings of Infotech State of the Art Conference: Why Software Projects Fail,* Pergamon Press, pp. 11.1–11.25, April 1978. (Reprinted in [Lehman and Belady 1985], pp. 247–274.)

[Lehman 1980] Lehman, M.M., "Programs, Life Cycles and Laws of Software Evolution", *Proceedings of the IEEE Special Issue on Software Engineering*, Vol. 68, No. 9, pp. 1060–1076, September 1980.

[Lehman 1981] Lehman, M.M., "Programming Productivity – A Life Cycle Concept", *Proceedings CompCon 81, IEEE Catalogue No. 81CH–1702–0,* pp. 232–241, September 1981.

[Lehman and Belady 1985] Lehman, M.M. and Belady, L., *Program Evolution, Processes of Software Change*, A.P.I.C. Studies in Data Processing No. 27, Academic Press, London, 1985.

[Lerner and Habermann 1990] Lerner, B.S. and Habermann, A.N., "Beyond Schema Evolution to Database Reorganisation", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications,* pp. 67–76, October 1990.

[Levin *et al.* 1992] Levin, R., McJones, P.R., Ayers, R.M., Brown, M.R., Chiu, S.Y., Ellis, J.R. and Hanna, C.B., Precise Configuration and Construction of Large Software Systems using Vesta (Working Draft), Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, USA, August 1992.

[Lieberherr and Holland 1989] Lieberherr. K.J. and Holland, I.M., "Tools for Preventive Software Maintenance", *Proceedings of Conference on Software Maintenance (Miami, FL, USA, 16th–19th October 1989),* pp. 2–13, IEEE Computer Society Press, Los Alamitos, CA, 1989.

[Lientz and Swanson 1981] Lientz, B.P. and Swanson, E.B., "Problems in Application Software Maintenance", *Communications of the ACM*, Vol. 24, No. 11, pp. 764–769, November 1981.

[Lientz *et al.* 1978] Lientz, B.P., Swanson, E.B. and Tompkins, G.E., "Characteristics of Application Software Maintenance", *Communications of the ACM*, Vol. 21, No. 6, pp. 466–471, June 1978.

[Loboz 1989] Loboz, Z., "Monitoring Execution of PS-algol Programs", In *Persistent Object Stores (Proceedings of the Third International Workshop, 10th–13th January 1989, Newcastle, New South Wales, Australia),* Rosenberg, J. and Koch, D. (editors), Springer-Verlag and British Computer Society, pp. 279–288, 1989.

[Lopes 1993] Lopes, J.C., ShTh – Show Thesaurus User Interface, Technical report in preparation, Computing Science Department, University of Glasgow, 1993.

[Maes 1987] Maes, P., "Concepts and Experiments in Computational Reflection", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (Orlando, FL, 4th–8th October 1987),* 1987.

[Marche 1993] Marche, S., "Measuring the Stability of Data Models", *European Journal on Information Systems*, Vol. 2, No. 1, pp. 37–47, 1993.

[Marti 1983] Marti, R.W., "Integrating Database and Program Descriptions using an ER-Data Dictionary", In *Database Techniques for Professional Workstations*, Zehnder, C.A. (editor), pp. 119–140, ETH, Zürich, September 1983.

[Matthes *et al.* 1992] Matthes, F., Rudloff, A., Schmidt, J.W. and Subieta, K., The Database Programming Language DBPL User and System Manual, Technical Report FIDE/92/47, ESPRIT Basic Research Action, Project Number 3070 – FIDE, 1992.

[McKenzie and Snodgrass 1990] McKenzie, E. and Snodgrass, R., "Schema Evolution and the Relational Algebra", *Information Systems*, Vol. 15, No. 2, pp. 207–232, 1990.

[Meekel and Viala 1988] Meekel, J. and Viala, M., "LOGISCOPE: A Tool for Maintenance", *Proceedings of Conference on Software Maintenance (Phoenix, AR, USA, 24th–27th October 1988)*, pp. 328–334, IEEE Computer Society Press, Los Alamitos, CA, 1988.

[Members 1990] Members of the FIDE types club with Atkinson, M.P. and Richard, P. as editors, Types for Large Scale Systems, Club Report of Meeting in Pisa, 5th–6th July, 1990, Technical Report FIDE/90/1, ESPRIT Basic Research Action, Project Number 3070 – FIDE, October 1990.

[Meyers *et al.* 1993] Meyers, S., Duby, C.K. and Reiss, S.P., "Constraining the Structure and Style of Object-Oriented Programs", *Proceedings of the First Workshop on Principles and Practice of Constraint Programming (PPCP93)*, April 1993. Also available as Brown University Computer Science Department Technical Report CS-93-12, April 1993.

[Milner 1984] Milner, R., "A Proposal for Standard ML", *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming (Austin, Texas, August 1984)*, pp. 184–197, ACM, New York, 1984.

[Mitchell and Plotkin 1985] Mitchell, J.C. and Plotkin, G.D., "Abstract Types Have Existential Types", *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pp. 37–51, New Orleans, January 1985.

[Monk and Sommerville 1993] Monk, S. and Sommerville, I., "Schema Evolution in OODBs Using Class Versioning", SIGMOD Record, Vol. 22, No. 3, pp. 16–22, September 1993.

[Morrison *et al.* 1989a] Morrison, R., Brown, F., Connor, R. and Dearle, A., The Napier88 Reference Manual, Research Report PPRR-77-89, Universities of Glasgow and St Andrews, 1989.

[Morrison *et al.* 1989b] Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. and Atkinson, M.P., "The Napier Type System", In *Persistent Object Stores (Proceedings of the Third International Workshop, 10th–13th January 1989, Newcastle, New South Wales, Australia)*, Rosenberg, J. and Koch, D. (editors), Springer-Verlag and British Computer Society, pp. 3–18, 1989.

[Morrison *et al.* 1990] Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P., "On the Classification of Binding Mechanisms", *Information Processing Letters*, Vol. 34, No. 1, pp. 51–55, February 1990.

[Munro 1993] Munro, D., On the Integration of Persistence, Concurrency and Distribution, PhD thesis in preparation, Department of Mathematical and Computational Sciences, University of St Andrews, 1993.

[Nakagawa and Futatsugi 1991] Nakagawa, A.T. and Futatsugi, K., "Propagating Changes in Algebraic Specifications", *Software Engineering Journal*, Vol. 6, No. 6, pp. 476–486, November 1991.

[Nelson 1992] Nelson, R.J, *Naming and Reference*, In series *The Problems of Philosophy*, Routledge, London 1992.

[O'Brien *et al.* 1987] O'Brien, P.D., Halbert, D.C. and Kilian, M.F., "The Trellis Programming Environment", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (Orlando, FL, 4th–8th October 1987)*, pp. 91–102, 1987.

[Ohori *et al.* 1989] Ohori, A., Buneman, O.P. and Breazu-Tannen, V., "Database Programming in Machiavelli – a Polymorphic Language with Static Type Inference", *Proceedings of the ACM SIGMOD 1989 Conference on the Management of Data (Portland, Oregon, 31st May – 2nd June), SIGMOD Record*, Vol. 18, No. 2, pp. 424–433, June 1989.

[Olle and Black 1988] Olle, W. and Black, M., "Data Levels in IRDS", In *The Future of Data Dictionaries, DATABASE 88 (19th–20th May 1988, Open University, Milton Keynes)*, Holloway, S. (editor), pp. 31–48, Gower Technical, The British Computer Society Database Specialist Group, 1988.

[Osborn 1989] Osborn, S.L., "The Role of Polymorphism in Schema Evolution in an Object-Oriented Database", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 3, pp. 310–317, September 1989.

[Osterweil 1987] Osterweil, L.J., "Software Processes are Software Too", *Proceedings of the Ninth International Conference on Software Engineering*, March 1987.

[Osterweil and Fosdick 1976] Osterweil, L.J. and Fosdick, L.D., "DAVE – A Validation, Error Detection and Documentation System for FORTRAN Programs", *Software – Practice and Experience*, Vol. 6, No. 4, pp. 473–486, 1976.

[Oxford 1961] *The Oxford English Dictionary,* Oxford University Press, London, 1961.

[Panel 1989] Panel on Schema Evolution and Version Management, Object-Oriented Database Workshop in OOPSLA'88, *SIGMOD Record,* Vol. 18, No. 3, pp. 90–95, September 1989.

[Parikh and Zvegintsov 1983] Parikh and Zvegintsov, "The World of Software Maintenance", *Tutorial on Software Maintenance*, Parikh and Zvegintsov (editors), Computer Society Press, Los Alamitos, CA, 1983.

[Parnas 1972] Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, Vol. 15, No. 12, pp. 1053–1058, December 1972.

[Parsys 1993] FTK – A Fortran Toolkit, Parsys, 1993. (See article in *Engineering Computing Newsletter SERC*, Rutherford Appleton Laboratory, Vol. 44, pp. 2–3, May 1993.)

[Penney and Stein 1987] Penney, D.J. and J. Stein, "Class Modification in the GemStone Object-Oriented DBMS", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications,* pp. 111–117, October 1987.

[Pfleeger 1987] Pfleeger, S.L., *Software Engineering – The Production of Quality Software*, Macmillan, 1987.

[Pressman 1992] Pressman, R.S., *Software Engineering – A Practitioner's Approach,* Third edition, McGraw-Hill, 1992.

[PS-algol 1987] PS-algol Reference Manual, Fourth edition, Research Report PPRR-12-87, Universities of Glasgow and St Andrews, 1987.

[PSL 1992] Polyhedra: Application Generation Environment, Version 1.0 (Beta 4) Release, Perihelion Software Ltd., December 1992.

[Putnam 1982] Putnam, L.H., "Software Cost Estimating and Life Cycle Control", *IEEE Catalog*, 1982.

[Qin 1993] Qin, Z., Second Year Report, Computing Science Department, University of Glasgow, 1992.

[Quong and Linton 1991] R.W. Quong and M.A. Linton, "Linking Programs Incrementally", *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, pp. 1–20, January 1991.

[Reps and Teitelbaum 1989] Reps T. W. and Teitelbaum T., *The Synthesizer Generator: A System for Constructing Language-Based Editors*, In series *Texts and Monographs in Computer Science*, Springer-Verlag, 1989.

[Ritchie *et al.* 1978] Ritchie, D.M., Johnson, S.C., Lesk, M.E. and Kernighan, B.W., "The C Programming Language", *Bell Systems Technical Journal*, Vol. 57, No. 6, pp. 1991–2020, 1978.

[Rochkind 1975] Rochkind, M.J., "The Source Code Control System", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 4, pp. 364–370, December 1975.

[Roddick 1992] Roddick, J.F., "SQL/SE – A Query Language Extension for Databases Supporting Schema Evolution", *SIGMOD Record,* Vol. 21, No. 3, pp. 10–16, September 1992.

[Royce 1970] Royce, W.W., "Managing the Development of Large Software Systems", *Proceedings of IEEE WESCON*, August 1970.

[Ryder 1979] Ryder, B.G., "Constructing the Call Graph of a Program", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, pp. 216–226, 1979.

[Saal and Weiss 1977] Saal, H.J. and Weiss, Z., "An Empirical Study of APL Programs", *Computer Languages*, Vol. 2, No. 3, pp. 47–60, 1977.

[Schefström 1991] Schefström, D., "The Arcs Experience", *Proceedings of Third European Software Engineering Conference (Milan, Italy, October 1991),* Lamsweerde A. van and Fugetta A. (editors), pp. 443–464, Lecture Notes in Computer Science 550, Springer-Verlag, 1991.

[Schmidt 1977] Schmidt, J.W., "Some High Level Language Constructs for Data of Type Relation", *ACM Transactions on Database Systems*, Vol. 2, No. 3, pp. 247–261, September 1977.

[Schmidt and Matthes 1992] Schmidt, J.W. and Matthes, F., The Database Programming Language DBPL Rationale and Report, Technical Report FIDE/92/46, ESPRIT Basic Research Action, Project Number 3070 – FIDE, 1992.

[Schwanke and Kaiser 1988] Schwanke, R.W. and Kaiser, G.E., "Smarter Recompilation", *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 4, pp. 627–632, October 1988.

[Schwanke and Platoff 1989] Schwanke, R.W. and Platoff, M.A., "Cross References are Features", *Proceedings Second International Workshop on Software Configuration Management (Princeton, New Jersey, November 1989),* Published as *Software Engineering Notices*, pp. 86–95, 1989.

[Sheard 1990] Sheard, T., A User's Guide to TRPL: a Compile-Time Reflective Programming Language, Dept. of Mathematics and Computer Science, Amherst College, Amherst, Ma 01002, USA, September 1990.

[Sheard 1991] Sheard, T., "Automatic Generation and Use of Abstract Structure Operators", *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 531–557, 1991.

[Sheard and Stemple 1989] Sheard, T. and Stemple, D., "Automatic Verification of Database Transaction Safety", *ACM Transactions on Database Systems*, Vol. 14, No. 3, pp. 322–368, September 1989.

[Shepard *et al.* 1992] Shepard, T., Sibbald, S. and Wortley, C., "A Visual Software Process Language", *Communications of the ACM*, Vol. 35, No. 4, pp. 37–44, April 1992.

[Sjøberg 1991] Sjøberg, D., The Thesaurus – A Tool for Meta Data Management, Technical Report FIDE/91/6, ESPRIT Basic Research Action, Project Number 3070 – FIDE, February 1991.

[Sjøberg 1992] Sjøberg, D., Measuring Name and Identifier Usage in Napier88 Applications, Technical Report FIDE/92/37, ESPRIT Basic Research Action, Project Number 3070 – FIDE, 1992.

[Sjøberg 1993] Sjøberg, D., "Quantifying Schema Evolution", *Information and Software Technology*, Vol. 35, No. 1, pp. 35–44, January 1993.

[Sjøberg *et al.* 1993] Sjøberg, D., Atkinson, M.P., Lopes, J. and Trinder, P., "Building an Integrated Persistent Application", *Fourth International Workshop on Database Programming Languages (30th August – 1st September, Manhattan, New York City, USA),* Springer-Verlag, 1993.

[Skarra and Zdonik 1987] Skarra, A.H. and Zdonik, S.B., "Type Evolution in an Object-Oriented Database", In *Research Directions in Object-Oriented Programming,* Shriver, B.S. and Wegner, P. (editors), pp. 393–415, MITP, Cambridge, MA, Computer Systems, 1987.

[Sockut and Goldberg 1979] Sockut, G.H. and Goldberg, R.P., "Database Reorganization – Principles and Practice", *ACM Computing Surveys,* Vol. 11, No. 4, pp. 371–395, December 1979.

[SoftwareAG 1990] The Predict Reference Manual Version 3.1, PRD-311-030, Software AG, Germany, 1990.

[Sommerville 1992] Sommerville, I., *Software Engineering*, Fourth edition, Addison Wesley, 1992.

[Sommerville 1993] Sommerville, I., Cooperative Systems Engineering, Seminar, University of Glasgow, March 1993.

[Sommerville and Morrison 1987] Sommerville. I. and Morrison, R., *Software Development with Ada*, Wokingham: Addison-Wesley, 1986.

[Spurr 1988] Spurr, K., "Introduction to the ISO IRDS Standards", In *The Future of Data Dictionaries, DATABASE 88 (19th–20th May 1988, Open University, Milton Keynes)*, Holloway, S. (editor), pp. 7–18, Gower Technical, The British Computer Society Database Specialist Group, 1988.

[Stemple 1989] Stemple, D., "Exploiting the Potential of Persistent Object Stores", In *Persistent Object Stores (Proceedings of the Third International Workshop, 10th–13th January 1989, Newcastle, New South Wales, Australia)*, Rosenberg, J. and Koch, D. (editors), pp. 45–55, Springer-Verlag and British Computer Society, 1989.

[Stemple *et al.* 1992] Stemple, D., Stanton, R.B., Sheard, T., Philbrow, P.C., Morrison, R., Kirby, G.N.C., Fegaras, L., Cooper, R.L., Connor, R.C.H., Atkinson, M.P. and Alagic, S., Type-Safe Linguistic Reflection: A Generator Technology, Technical Report FIDE/92/49, ESPRIT Basic Research Action, Project Number 3070 – FIDE, 1992.

[Strachey 1967] Strachey, C., *Fundamental Concepts in Programming Languages*, Oxford University Press, Oxford, 1967.

[Sun Microsystems 1988a] The Sun Operating System Release 4.1, Sun Microsystems, October 1988.

[Sun Microsystems 1988b] Introduction to the NSE™, Part No: 800-2362-10 (Draft 7 March 1988), Sun Microsystems, 1988.

[Sutton *et al.* 1990] Sutton, S.M., Heimbigner, D. and Osterweil, L.J., "Language Constructs for Managing Change in Process-Centered Environments", *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pp. 206–217, December 1990.

[Swanson 1976] Swanson, E.B., "The Dimension of Maintenance", *Proceedings of the Second International Conference on Software Engineering*, pp. 492–497, October 1976.

[Symantec 1989] THINK C™ User's Manual, Symantec Corporation, 1989.

[Tabkha 1991] Tabkha, I., "An Implementation of the Parts Explosion Problem", Second Annual FIDE Review Meeting, Computing Science Department, University of Glasgow, September 1991.

[Tabkha 1993] Tabkha, I., Two Implementations of Parameterised Abstract Data Types, Technical report in preparation, University of Glasgow, 1993.

[Tarr and Clarke 1993] Tarr, P. and Clarke, L.A., "PLEIADES: An Object Management System for Software Engineering Environments", to appear in *ACM SIGSOFT '93: Proceedings of the Symposium of the Foundations of Software Engineering*, Los Angeles, CA, December 1993. Also available as University of Massachusetts, Amherst, Computer Science Department CMPSCI Technical Report 93-64, July 1993.

[Teitelman and Masinter 1981] Teitelman, W. and Masinter, L., "The Interlisp Programming Environment", *IEEE Computer*, Vol. 14, No. 4, pp. 25–33, April 1981.

[Thompson 1992] Thompson, A.K., "CASE Data Integration: The Emerging International Standards", *ICL Technical Journal*, Vol. 8, No. 1, pp. 54–66, May 1992.

[Tichy 1985] Tichy, W.F., "RCS – A System for Version Control", *Software – Practice and Experience*, Vol. 15, No. 7, pp. 637–654, July 1985.

[Tichy 1986] Tichy, W., "Smart Recompilation", *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 3, pp. 273–291, July 1986.

[Tresch and Scholl 1993] Tresch, M. and Scholl, M.H., "Schema Transformation without Database Reorganisation", *SIGMOD Record*, Vol. 22, No. 1, pp. 21–27, March 1993.

[Trinder 1991] Trinder, P.W., "Comprehensions, a Query Notation for DBPLs", *Proceedings of the Third International Workshop on Database Programming Language (Nafplion, Greece, 27th–30th August 1991)*, Kanellakis, P. and Schmidt, J.W. (editors), pp. 55–70, Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Tsichritzis and Lochovsky 1982] Tsichritzis, D.C. and Lochovsky, F.H., *Data Models*, Englewood Cliffs, N.J. Prentice-Hall, 1982.

[Uhrowczik 1973] Uhrowczik, P.P., "Data Dictionary/Directories", *IBM Systems Journal*, Vol. 12, No. 4, pp. 332–350, 1973.

[Waller 1991] Waller, E., "Schema Updates and Consistency", *Proceedings of the International Conference on Deductive and Object-Oriented Databases (DOOD) (Munich, Germany, 16th–18th December 1991)*, pp. 167–188, Lecture Notes in Computer Science 566, Springer-Verlag, 1991.

[Webster 1961] *Webster's Third New International Dictionary of the English Language Unabridged*, editor in chief P.B. Gove and the Merriam-Webster editorial staff, G. & C. Merriam Co., G. Bell & Sons Ltd., London, 1961.

[Wegner and Zdonik 1988] Wegner, P. and Zdonik, S.B., "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like", *Proceedings of the European Conference on Object-Oriented Programming (Oslo, 15th–17th August 1988),* Gjessing, S. and Nygaard, K. (editors), pp. 55–77, Lecture Notes in Computer Science 322, Springer-Verlag, 1988.

[Weiser 1982] Weiser, M., "Programmers Use Slices When Debugging", *Communications of the ACM*, Vol. 25, No. 7, pp. 446–452, July 1982.

[Weiser and Shneiderman 1987] Weiser, M. and Shneiderman, B., "Human Factors of Computer Programming", In *Handbook of Human Factors*, Salvendy, G. (editor), pp. 1398–1415, John Wiley & Sons, 1987.

[Wiederhold *et al.* 1992] Wiederhold, G., Wegner, P. and Ceri, S., "Toward Megaprogramming", *Communications of the ACM*, Vol. 35, No. 11, pp. 89–99, November 1992.

[Wolf *et al.* 1989] Wolf, A.L., Clarke, L.A. and Wileden, J.C., "The AdaPIC Tool Set: Supporting Interface Control and Analysis Throughout the Software Development Process", *IEEE Transactions on Software Engineering,* Vol. SE-15, No. 3, pp. 250–263, March 1989.

[Zelkowitz 1978] Zelkowitz, M.V., "Perspectives on Software Engineering", *ACM Computing Surveys,* Vol. 10, No. 2, pp. 197–216, June 1978.

[Zicari 1992] Zicari, R., "A Framework for Schema Updates in an Object-Oriented Databases System", In *Building an Object-Oriented Database System: The Story of O_2*, Bancilhon, F., Delobel, C. and Kanellakis, P. (editors), pp. 146–182, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

Index

Acheampong (1993) 92
AD/Cycle 62
Ada 116, 180
Adams *et al.* (1989) 56
AdaPIC 180
Agresti and Evanco (1992) 9, 92
Ahlsen *et al.* (1983) 53
Albano (1983) 54
Albano *et al.* (1985) 72, 77
Allen *et al.* (1982) 59
ANSI (1988) 59
application model 51, 117
application-program 121
APSE 64, 116
Archer and Devlin (1986) 64
Arcs 64
Ariav (1991) 51
Atkinson (1978) 66
Atkinson (1989) 68
Atkinson (1990) 33
Atkinson (1992) 68, 87
Atkinson (1993) 72–73, 106, 117, 144, 178
Atkinson and Buneman (1987) 68, 72
Atkinson and Morrison (1985) 66, 68–69
Atkinson and Morrison (1986) 71
Atkinson *et al.* (1982) 7, 66
Atkinson *et al.* (1983a) 7, 66, 68
Atkinson *et al.* (1983b) 66
Atkinson *et al.* (1983c) 66
Atkinson *et al.* (1988) 68, 71, 167
Atkinson *et al.* (1990) 77, 87, 161
Atkinson *et al.* (1991a) 77, 92
Atkinson *et al.* (1991b) 77
Atkinson *et al.* (1993) 76, 134, 141
awk 25

Bachman (1988) 48
Bailey (1989) 92
Banerjee *et al.* (1987) 2, 33, 51–52, 53
Barclay *et al.* (1992) 92
Barnard *et al.* (1982) 5
Batini *et al.* (1986) 51
Baxter (1992) 48
Berman (1991) 77
binding 71
  categories 121–122
  export 121
  import 121
  internal 121
  L-value 71, 74, 119, 140
  R-value 71
  unused 180
Birnie (1991) 92
Bjørner (1991) 48
block
  depth 80
  sequence 80
Boehm (1988) 45
Bott (1989) 64
Bourne (1979) 16, 62, 89
Bratsberg (1993) 51
Brodie (1992) 48
Brooks (1975) 4
Brown (1989) 68
Brunhoff (1991) 56
build management 155
Buxton (1980) 64

C 14, 56–57
C++ 180
call-graphs 61
Cardelli (1989a) 178
Cardelli (1989b) 69
Cardelli and Wegner (1985) 68
Cartmell and Alderson (1989) 50
Casais (1991) 51
CCEL 180
CDIF 60
change
  causes 2–3, 169, 171
  control 49
  history 24, 31
  management 5, 49–50
  measurements 7
  process 49
  propagation 4, 23, 36, 106, 179
    measurements 109
  schema (see schema evolution)
Chapin (1988) 46
Chen (1976) 124
Chikofsky and Cross (1990) 1
class evolution 51
Clifton (1990) 14
closure (see procedure)
COBOL 57, 116
cohesion 2

Colbrook and Smythe (1989) 48
Collofello and Buck (1987) 4
compilation (see also EnvMake)
  separate 72
  smart 56, 160
  support 57
comprehension query language 84
configuration management 55, 179
Connor (1991) 68, 73, 81, 106, 117, 158
Connor *et al.* (1990) 144
Connor *et al.* (1991) 54
consistency 50 (see also SPASM)
consistency checking 24
Constantine and Yourdon (1979) 2, 116
constraints 10, 118, 173 (see also SPASM)
  specification 180
  violation 152
context 80
conversion 53
Cooper (1990a) 68
Cooper (1990b) 178
Cooper and Qin (1992) 51, 178
Copeland and Maier (1984) 66
coupling 2
cpp 25
crane 8
cross-referencers 58, 61
Cutts (1993a) 10, 69, 76, 87, 92, 106, 117, 179
Cutts (1993b) 77
Cutts *et al.* (1990) 76, 84, 92, 159

Dahl *et al.* (1972) 2, 116
Dart (1991) 57
Dart *et al.* (1987) 63
Data Dictionary Systems Working Party 59
data dictionary 5, 58, 61–62
data dictionary tools 89, 179
data flow analysis 57
data modelling 48, 178
database
  programming 66
  reorganisation 22, 53
Date (1990) 116
datum 15
Davie and Morrison (1981) 77
DBPL 77, 89
DBPLXref 89
DDS 62
Dearle (1987) 106, 117
Dearle (1988) 68, 70
Dearle *et al.* (1989) 68
Dearle *et al.* (1992) 106, 117
DEC (1989) 16, 58, 89
DEC (1993) 58, 85
declaration and use 18–19, 24, 127, 152
DeMarco (1979) 118
dependency 56, 89 (see also EnvMake)
DeRemer and Kron (1976) 3
diff 25
Display Language 14
documentation 4
  automatic 48
Dolk and Kirsch (1987) 60
Dolotta *et al.* (1978) 63
drop-clause 131
drop-program 120
DSEE 56

Eclipse 64
ECMA (1990) 60
EIA (1991) 60
Elshoff (1976) 9, 92
England and Selwyn (1990) 14
Entity-Relationship diagram 124
environments (Napier88) 70, 73
  aliases 165
  assignments 165
  contains-check 70
  drop-clause 70
  identification 162
  insert-declaration 70
  modification 142
  references to 181
  returned 163
  use-clause 70
EnvMake 10, 147, 166
  application structure 148
  build management 155
  compilation 158, 161
  dependency matrix 150
  dependency table 148
  execution 160–161
  experiences 154
  flexibility 154, 168
  implementation 161
  installation 160
  menu 149
  ordering 153
  performance 162
  plan 157
  status information 156
  up-to-date information 161
evolution 46
  laws of 3
  schema (see schema evolution)

execution 160
    order 162
experimentation 185
explode 85

Farkas *et al.* (1992) 76, 183
Fegaras and Stemple (1991) 51
Fegaras *et al.* (1989) 51
Feldman (1979) 10, 50, 55
Feldman (1991) 180
Ferraby (1991) 3, 49
formal specifications 48
FORTRAN 57, 116
Fosdick and Osterweil (1976) 57
FUSE 58, 85

Galileo 72, 77
Gandalf 65
Gløersen (1993) 43
Goldberg (1984) 65
Gopal *et al.* (1992) 58
granularity 31, 50, 138, 158
Greenwood *et al.* (1992) 68
grep 25, 89
Griswold and Notkin (1992) 49

Habermann and Notkin (1986) 65
Hippo language 14
HMS
    execution log 187
    system 14
Holloway (1988a) 59
Holloway (1988b) 61–62, 179
Holt (1993) 186
Humphrey (1989) 49
hyper-programming 76, 181–183
    button 184
    constraints 183
    gesture-based 183
    naming 184
hyper-references 142, 166, 179, 181
hyper-world 183

IBM (1978) 67, 110
IBM (1980) 16, 59
IBM (1990) 59, 62, 176
IBM (1991) 62, 176
IBM (1992) 49
identifier 90
    information 5
identity 162
Imber (1991) 60
impact analysis 23
impedance mismatch 66
implode 85
incremental construction 72
insert-program 120–121
installation 160
integrity rules 116
intellectual capacity 180, 186
Interlisp 64
IPSE 63
IRDS 59
ISO (1990) 59–60

Jackson (1975) 2, 116
Jackson (1983) 118
Jacobs and Hull (1991) 139
Jandrasics (1981) 48

Kay (1992) 60
Keables *et al.* (1988) 57
Kim and Chou (1988) 33, 51
kind 80
King (1967) 58
King (1969) 58
Kirby (1993) 68, 70, 76–77, 183
Kirby (1993) 182
Kirby and Dearle (1990) 76–77, 87, 89
Kirby *et al.* (1992) 182
Knuth (1972) 9, 92
Knuth (1973) 131–132
Krueger (1992) 4

L-values (see binding)
language designer 91
Leblang *et al.* (1985) 56
Lehman (1976) 7
Lehman (1978) 48
Lehman (1980) 4, 46
Lehman (1981) 1
Lehman and Belady (1985) 3, 6
Lerner and Habermann (1990) 2, 33, 51
level of detail (see granularity)
Levin *et al.* (1992) 57
Lieberherr and Holland (1989) 51
Lientz and Swanson (1981) 3
Lientz *et al.* (1978) 1, 3, 6
life cycle 46
link 184 (see also hyper-references)
Loboz (1989) 92
Lopes (1993) 84

Machiavelli 160
Maes (1987) 77
maintenance 4, 118

adaptive 1
corrective 1
methodology 10, 138
perfective 1
phase 46
proportion 1
Make 55, 147
makedepend 56
Makefile 55, 147, 155
maps 77, 85, 87
Marche (1993) 42, 171, 185
Marti (1983) 61, 86
Matthes *et al.* (1992) 89
McKenzie and Snodgrass (1990) 51
measurements 6, 184–185
Ada 92
APL 92
constancy 98–99
context 98, 190
declaration and use 128
environments 110–111, 164, 192, 193
FORTRAN 92
kind 95, 189
lines of code 93
name length 100–101
name use 94
persistent programming languages 92
PL/1 92
procedures 105–106
context 107
polymorphic 108–109
specialised 108–109
program categories 125
size 92
structure fields 104
transfer secondary storage 110
type definitions 101–103, 126, 191
usage 96
use-clauses 129
variant tags 104
Meekel and Viala (1988) 48
mega-programming 3
Members (1990) 73
meta-data 58
meta-database 58
methodology 10, 138, 185 (see also maintenance, SPASM)
programming 115
system development 115, 118
Meyers *et al.* (1993) 180
Milner (1984) 69
mismatch 89
Mitchell and Plotkin (1985) 68
Monk and Sommerville (1993) 51
Morrison *et al.* (1989a) 7, 68
Morrison *et al.* (1989b) 68
Morrison *et al.* (1990) 68, 70, 72
Munro (1993) 76, 127, 131, 181

Nakagawa and Futatsugi (1991) 48
name 80, 90 (see also measurements)
information 5
meaning 5
occurrence 90
naming 162
convention 133–134
Napier88 9, 68, 73
browser 87, 89
language processing technology 77
libraries 141
programming environment 76
type system 68
Napier88-in-Napier88 compiler 69, 77, 87, 179
Nelson (1992) 6
NinN (see Napier88-in-Napier88)

O'Brien *et al.* (1987) 64
Ohori *et al.* (1989) 160
Olle and Black (1988) 59
Osborn (1989) 51
Osterweil (1987) 49
Osterweil and Fosdick (1976) 57
Oxford (1961) 5

P-Pascal 77
Panel (1989) 33, 51
Parikh and Zvegintsov (1983) 1
Parnas (1972) 2
Parsys (1993) 57
partial order 131–132
PAS 3
Pascal-R 77
PCTE 60
Penney and Stein (1987) 33, 51
performance 86
trends 186
persistence
independence 67
orthogonality 67
reachability 68
research 68
Persistent Application System (see PAS)
persistent location binding methodology 117, 119, 180
persistent programming 7, 66

environments 177, 185
persistent store 73, 153, 174
Pfleeger (1987) 1
PLEIADES 180
Polyhedra 14
polymophism (see procedures)
Predict 62, 85
Pressman (1992) 46, 49
procedure
change 139, 142, 165
closure 69, 70, 75
higher-order 69, 138
polymorphic 68, 77 (see also measurements)
program categories 120, 152, 162
program management 165
program slicing 57
programming
culture 117
languages 185
process 63, 146, 168
programming-in-the-large 3
project management 49
prototyping 2
PS-algol (1987) 68
Putnam (1982) 1

Qin (1993) 76
query dictionary 15, 19, 36
Quong and Linton (1991) 56

RCS 55
recompilation (see compilation)
reflection 77
Repository Manager 59, 62
repository 59, 62, 176
Reps and Teitelbaum (1989) 65
reuse 4, 87
reverse engineering 48
Ringad comprehensions 85
ripple effect 4
Ritchie *et al.* (1978) 57
Rochkind (1975) 55
Roddick (1992) 51
Royce (1970) 45
Ryder (1979) 48

Saal and Weiss (1977) 9, 92
scale 3, 44
scanning environments 86
SCCS 55
Schefström (1991) 64
schema
change (see schema evolution)
in Napier88 143
schema evolution 42, 139, 143, 178
categories 22
consequences 33, 36, 38, 52, 144
filtering 53
object-oriented databases 51
quantification 7, 34–35, 43, 171
problems 40
research 51, 54
Schmidt (1977) 77
Schmidt and Matthes (1992) 77, 89
Schwanke and Kaiser (1988) 56
Schwanke and Platoff (1989) 56, 156
sed 25
Sheard (1990) 51
Sheard (1991) 180
Sheard and Stemple (1989) 180
Shepard *et al.* (1992) 49
ShTh 84
Sjøberg (1991) 6, 21
Sjøberg (1992) 98, 101, 104–105, 129, 164
Sjøberg (1993) 7, 14, 52, 185
Sjøberg *et al.* (1993) 7, 9, 79, 85–86, 172
Skarra and Zdonik (1987) 2, 33, 51, 53
Smalltalk 65
Sockut and Goldberg (1979) 53
software
development process 45
evolution (see evolution)
life cycle 45, 49, 59, 63, 90
process modelling 49
SoftwareAG (1990) 16, 62, 85–86, 89
Sommerville (1992) 46
Sommerville (1993) 64
Sommerville and Morrison (1987) 64, 116
source code analysers 89
SPASM 10, 115, 118, 165
checking 151–152
constraints 122–124, 136, 180
extensibility 180
partial order 133
persistent store 135
violation 154, 167
spiral model 45
Spurr (1988) 59
SQL 15, 26, 90
Startup-program 120–121
static program analysis 57
Stemple (1989) 180
Stemple *et al.* (1992) 77
Stoneman report 64
Strachey (1967) 69

Structure-oriented environments 65
structured programming 116
structuring conventions 133
stub methodology (see persistent location binding methodology)
stubs 119, 129, 153
subtyping 54
Sun Microsystems (1988a) 25
Sun Microsystems (1988b) 57
support environment
  language independent 63
  language specific 64
supporting tools 2, 7, 45 (see also thesaurus tools, EnvMake)
Sutton *et al.* (1990) 49
Swanson (1976) 1
Symantec (1989) 56, 167
Synthesizer Generator 65
system catalogue 62

Tabkha (1991) 92
Tabkha (1993) 92
Tarr and Clarke (1993) 180
Teitelman and Masinter (1981) 64
thesaurus (definition) 5
thesaurus tools
  comparison 88
  HMS 6, 13, 15
    implementation 25
    Query_Dictionary relation 19
    thesaurus interface 20
    Thesaurus relation 16
    Versions_Thesaurus relation 20, 31
  Napier88 9, 80
    definition 82, 87
    entries 83
    implementation 87
    registration 85
    thesaurus interface 84
    thesaurus kernel 177
    thesaurus-based tools 172, 177
    update 86
THINK C 56
Thompson (1992) 61
Tichy (1985) 55
Tichy (1986) 56, 158
timestamping 158, 161
topological sorting 131–132
transaction 15
Trellis programming environment 64
Tresch and Scholl (1993) 53
Trinder (1991) 84–85, 172
Tsichritzis and Lochovsky (1982) 43
TSIT 79, 161 (see also thesaurus tools: Napier88)
type 68
  checking 71
  database 69, 159
  definitions 68, 125, 143, 152, 158
  dependency 158–159
  equivalence 68, 103, 125, 143, 179
  evolution 51 (see also schema evolution)
  identifier 68
  system 60, 68, 173, 178
type-program 120, 159

Uhrowczik (1973) 59
update functions 15
update-program 120–121
usage 80
use-clauses (see also environments)
  automatic generation 180–181

VAXset 58
version management 178
views 22, 36, 53
visualisation 150, 178

Waller (1991) 51, 139
waterfall model 45
Webster (1961) 10
Wegner and Zdonik (1988) 54
Weiser (1982) 57
Weiser and Shneiderman (1987) 5
Wiederhold *et al.* (1992) 3
WIN 76, 84, 92
Wolf *et al.* (1989) 180

Zelkowitz (1978) 1
Zicari (1992) 51