Versioning in Software Engineering Databases

Anund Lie

January 2, 1990 (reprint December 18, 1992)
Abstract

There is a great need for database support for software engineering and other CAD environments to replace the current, file-based, ad-hoc data repositories. However, many of the design goals and tradeoffs done in traditional database systems for data processing and information system applications are inappropriate for engineering databases. This thesis examines the database requirements posed by engineering applications, and surveys some of the recent work in the area.

One particular requirement to engineering databases is handling of multiple versions, both sequential revisions and parallel variants. The thesis discusses in greater detail versioning and its interaction with other features of engineering databases (data model, concurrency control etc.) We present an alternate model for versioning, the change-oriented model, which focuses on externally-visible functional properties rather than the individual versions.

The change-oriented model is used as a basis for versioning in the prototype EPOS database. The prototype implements an Entity-Relationship class data model, with (multiple) inheritance and a few other enhancements. Versioning is transparent and orthogonal to the data model. The prototype is used to implement a simple, versioned product archive for a software engineering environment. The product archive stores source code as well as product structure, derived objects, tool descriptions and derivation history. All work is performed in checked-out workspaces with normal Unix tools.
Contents

1 Introduction .......................................................... 9
  1.1 Software Engineering ......................................... 10
  1.2 The EPOS Project .............................................. 14

2 Engineering Databases .............................................. 16
  2.1 Requirements to the Data Model ............................ 17
    2.1.1 Classical data models ..................................... 18
    2.1.2 Data model features ...................................... 19
    2.1.3 Object-orientation ........................................ 24
    2.1.4 Schema evolution .......................................... 24
  2.2 Concurrency Control .......................................... 25
    2.2.1 Long transactions ......................................... 26
    2.2.2 Checkout and checkin ...................................... 28
    2.2.3 Optimistic concurrency control ......................... 29
    2.2.4 Nested transactions ...................................... 29
  2.3 Programming Language Interfaces ........................... 30
    2.3.1 Procedure-based data manipulation languages .......... 31
    2.3.2 Embedded data manipulation languages ................. 33
    2.3.3 Language extensions ...................................... 34
    2.3.4 Persistent programming languages ...................... 34

3 Versioning and configuration management ..................... 36
  3.1 Basic Versioning ................................................ 36
    3.1.1 Versioning concepts ...................................... 36
    3.1.2 Delta storage .............................................. 37
3.1.3 Applicability of version-oriented versioning ............... 38
3.1.4 Transparency of versioning .................................. 39
3.2 Configuration Management ........................................ 40
  3.2.1 Configuration selection .................................... 41
  3.2.2 Rebuild ..................................................... 41
  3.2.3 Special-purpose CM mechanisms ............................. 42
  3.2.4 Modularization ............................................. 43
  3.2.5 Distributed environments .................................. 44

4 Versioning and Configuration Management Systems .......... 46
  4.1 File-based systems ............................................. 46
    4.1.1 SCCS, RCS .............................................. 46
    4.1.2 Make .................................................... 48
    4.1.3 Cedar ................................................... 48
    4.1.4 DSEE ................................................... 49
    4.1.5 SIO ...................................................... 50
    4.1.6 Gandalf ................................................ 50
    4.1.7 Adele .................................................... 50
  4.2 Object Management Systems .................................... 52
    4.2.1 PCTE and PACT .......................................... 52
  4.3 Database systems ............................................... 53
    4.3.1 Damokles ................................................. 53
    4.3.2 Orion .................................................... 54

5 Change-Oriented Versioning ....................................... 55
  5.1 Introduction .................................................. 55
    5.1.1 Example ................................................ 57
  5.2 The Change-Oriented Model .................................... 59
    5.2.1 Basic versioning ....................................... 59
    5.2.2 Validity ............................................... 62
    5.2.3 The editing process ................................... 64
    5.2.4 Configuration descriptions ............................. 70
5.3 Synchronization and transactions ............................................ 75
  5.3.1 Synchronization in the choice space .................................. 75
  5.3.2 Overlapping ambitions .................................................... 76
  5.3.3 The notification service ................................................. 77
  5.3.4 Checkout/checkin ......................................................... 78
  5.3.5 Option-based synchronization ......................................... 81
  5.3.6 Primary ambition ........................................................ 82

5.4 Complexity of Change-Oriented Versioning ............................... 82
  5.4.1 Visibilities ............................................................... 83
  5.4.2 Checkout ................................................................. 85
  5.4.3 Ambitions ............................................................... 86
  5.4.4 Stability ............................................................... 87
  5.4.5 Intersecting visibilities .............................................. 88
  5.4.6 Validities .............................................................. 89
  5.4.7 Cutting down the complexity ....................................... 90

5.5 Comparison with Traditional Systems .................................... 91
  5.5.1 Emulating a version-oriented model .................................. 91
  5.5.2 Change-oriented vs. version-oriented versioning .................. 92

5.6 Application to Common Data Models .................................... 93
  5.6.1 Applicability of COV .................................................. 94
  5.6.2 Text files ............................................................. 94
  5.6.3 Relational model ..................................................... 95
  5.6.4 Binary data models ................................................... 99
  5.6.5 Entity-Relationship model ......................................... 100
  5.6.6 Object-oriented models ............................................ 102

6 The EPOS Database .......................................................... 103
  6.1 Data Model .............................................................. 103
    6.1.1 Entities ........................................................... 103
    6.1.2 Relations .......................................................... 104
    6.1.3 Complex objects .................................................. 105
    6.1.4 Attributes ........................................................ 105
6.1.5 Versioning ................................................. 106
6.1.6 Operations ................................................. 106
6.1.7 Methods ................................................... 108
6.1.8 Type refining ............................................ 109
6.2 Transactions ................................................. 109
  6.2.1 General properties ..................................... 109
  6.2.2 Copy-in and copy-out ................................. 110
  6.2.3 Low-level change propagation ....................... 111
6.3 Handling Derived Objects ................................. 111
6.4 Interfaces to the EPOS DB ................................. 112
  6.4.1 Notifications to the activity manager ............... 112

7 The EPOS Prototype ........................................ 114
  7.1 Overview of the Initial EPOSDB Prototype .......... 114
    7.1.1 Description ......................................... 114
    7.1.2 Functionality offered ............................... 115
    7.1.3 Database bridges .................................. 117
    7.1.4 Purpose of the prototype ......................... 117
  7.2 The Design of the EPOSDB Prototype .................. 118
    7.2.1 Product structure .................................. 118
    7.2.2 Tools and derivations .............................. 119
    7.2.3 Building the derivation graph .................... 120
    7.2.4 Tool modelling ..................................... 121
  7.3 The Prototype Database .................................. 124
    7.3.1 Versioning and transaction management for tables . 124
    7.3.2 Mapping the EPOS data model on tables ........... 126
    7.3.3 Consistency constraints and visibility update .... 129

8 Evaluation and Conclusions ................................ 133
  8.1 COV Evaluation ........................................... 133
    8.1.1 EPOSDB prototype .................................... 133
    8.1.2 Complexity issues ................................... 134
8.1.3 COV advantages ........................................... 135
8.2 Related Work .................................................. 136
  8.2.1 SCCS deltas ................................................ 136
  8.2.2 P-EDIT and MVPE ........................................... 136
  8.2.3 Miscellaneous ............................................... 137
8.3 Continued Work ................................................ 137

A Prolog Interface to EPOSDB Prototype 139
  A.1 General Principles .......................................... 139
  A.2 Database Query ............................................... 139
  A.3 Database Update ............................................. 140
  A.4 Working Context and Transaction Management .............. 141
  A.5 File Manipulation ........................................... 142
  A.6 Boolean Expressions ........................................ 142

B Schema for Database Prototype 144
  B.1 Data Definition Language ................................... 144
  B.2 Schema Listing ............................................... 146
    B.2.1 Entities ................................................. 146
    B.2.2 Relationships .......................................... 147
  B.3 Semantics of Schema ....................................... 148
    B.3.1 Composition structure .................................. 148
    B.3.2 Component and tool types ................................ 149
    B.3.3 Derivations ............................................. 150
List of Figures

5.1 Change-oriented vs. version-oriented versioning .......................................................... 56
5.2 Example validity ............................................................................................................... 58
5.3 Version selection in the change-oriented model ............................................................. 61
5.4 Fragments and visibilities ............................................................................................... 62
5.5 Edit task in the choice space .......................................................................................... 65
5.6 Visibility update example ............................................................................................... 67
5.7 Adding an option ............................................................................................................. 69
5.8 Implementation of orthogonal options .......................................................................... 71
5.9 Implementation of revision option .................................................................................. 72
5.10 Transactions with non-overlapping ambitions ............................................................... 76
5.11 Transactions with overlapping ambitions ...................................................................... 77
5.12 Compatibility matrix for transaction modes ................................................................... 80
5.13 Ambitions and primary ambitions ............................................................................... 82
5.14 Computing set operations on ambitions ..................................................................... 86
5.15 Example version graph ................................................................................................. 92

7.1 Product archive with workspaces .................................................................................. 116
7.2 Tool descriptions ........................................................................................................... 122
7.3 C compilation, before matcher ....................................................................................... 122
7.4 C compilation, after matcher .......................................................................................... 123
7.5 Representation of subtyping for entities ........................................................................ 127
7.6 Representation of subtyping for relationships .............................................................. 129
7.7 Database update algorithm ............................................................................................ 131

B.1 Derivations example ..................................................................................................... 150
List of Tables

5.1 COV glossary ............................................ 60
Acknowledgements

I’d like to thank my advisor, professor Reidar Conradi, for inviting me to a dr.ing. study, for believing in the work and prodding me on, and not least, for many stimulating technical discussions.

This work has been done in the context of the EPOS project, and the members of the project have been invaluable discussion partners. I’d like to thank them all, and mention in particular those who have been most involved with the database and configuration management work: Tor M. Didriksen, Svein O. Hallsteinsen, Per Holager, Even-André Karlsson. A very special thanks goes to Per Holager as the originator of the idea of change-oriented versioning, which I have only elaborated on.

I’d also like to thank the students I’ve acted as advisor for during the last year, Svein Erik Bratsberg and Erik Odberg for many discussions on databases which no doubt have served to clarify my thoughts, and Bjørn Munch and Christian Fabre for a lot of work spent implementing my designs.

I spent one year, 1987–1988, at Carnegie Mellon University with the Gandalf project. Thanks to the Gandalf group for many stimulating discussions. Even if the details are different, influences from the Gandalf database design can be seen in the transaction mechanism in EPOS.

During the period of my dr.ing. study, I have been supported partly by a dr.ing. fellowship from the University of Trondheim, partly by the EPOS project, which itself is funded by the Norwegian Council for Scientific and Industrial Research, grant ED0224.18457.
Chapter 1

Introduction

Development of large software systems is a challenging pursuit. The complexity of large systems is hard to handle, and large software systems are among the most complex systems ever built. No single person is able to keep an overview, and with large teams of cooperating persons, managing the communication is a challenge in itself.

In this situation, it is widely recognized that better software tools is one of the most important factors in attaining better productivity. We have the ordinary software development tools like editors, compilers and linkers, and somewhat more exotic tools for version archiving, minimizing recompilations, managing and running tests and so on. There are tools for documentation, simple text processors or more sophisticated systems integrating with the program development. In addition we have general "office infrastructure" tools, like electronic mail, spreadsheets, calendars etc. Finally, many development methodologies are tool-based, or are at least much more convenient to use if supported by tools.

To use this plethora of tools effectively, they have to be integrated in a uniform framework, a software development environment. Most importantly, it must be possible to exchange data between tools in a convenient way. Data formats must be compatible, or at least it must be possible to convert between them. User interface, command sets etc. must be as similar as possible. Pervasive functionality like configuration management and process management must probably be provided by the environment itself, and should be commonly and uniformly applicable.

Much work has been channeled into environment research, but there are still relatively few successful examples of highly integrated environments. The "straitjacketing effect" is infamous, and requires highly customizable environments to be overcome. One of the current trends is to use database technology to provide the "data integration" component of an environment. We are discussing this further in chapter 2, in particular the requirements this puts to the database, and to what extent these are fulfilled by current database technology.

One of these requirements is to be able to handle versioning and support configuration management: For software development, and engineering projects in general, the product and its components do generally occur in several versions. For one
thing, they evolve over time as development proceeds, and it is frequently import-
ent to record old revisions. But the product may also occur in different variants,
with different functional properties. Configuration management is basically prod-
uct version management and answers questions like: What components go into the
product? Which component versions are compatible and give a product with the
required properties? What version of the product was delivered to this customer?

Versioning and configuration management is discussed in chapter 3, both from a
database point of view and somewhat more in general. Chapter 4 illustrates this
by giving some examples, both of dedicated versioning and configuration manage-
ment systems, and of object management or database systems which incorporate
versioning.

In chapter 5, the change-oriented versioning model is presented, as an alternative to
the versioning model most existing systems use. We argue that this model has poten-
tial benefits, in particular with respect to parallel development, uniform versioning
of both contents and structure of a software system and for database versioning.

This versioning model is used as the basis for versioning in the EPOS database, the
central data repository in the EPOS software engineering environment. The overall
design of the EPOS data model is presented in chapter 6, while a detailed design for
a prototype implementation and application is found in chapter 7.

The remainder of the current chapter describes the context this work is set in.
Section 1.1 introduces the concept of software engineering, considering software de-
development to be an engineering discipline and treating it as such. Section 1.2 is a
brief overview of the software environment project this work is a part of.

1.1 Software Engineering

The concept “Software Engineering” is over twenty years old: It was coined at a
NATO conference in Garmisch-Partenkirchen in 1968. Already at that time, the
“Software Crisis” was plain to see, and software engineering was seen as the way to
meet this challenge. The extreme “softness” of software is treacherous:

- Flexibility and malleability.
- Lack of concrete, visible cost-carrying items (as for hardware).
- Lack of visibility for complexity and size.

These characteristics of software seem to seduce us to consistently aim for higher
functionality and complexity than we are actually able to implement. Even though
new tools and methodologies increase our capabilities, the appetite for new and more
complex software swells even faster.

The mounting awareness in the late sixties is that large-scale software development
should be an engineering discipline: Approaches from other, more mature engineer-
ing disciplines might be applicable to software development. However, after 20 years,
software engineering (SE) is still not a firmly established discipline. There is wide agreement about what some of the major components of SE are:

- Lifecycle or process models
- Methodology
- Tooling and tool integration
- Configuration management

The deeper one goes into the details of these, the less the agreement is—on content, conceptual framework, methodologies, terminology etc. A small summary of each is still felt to be in its place. Note that any classification, such as this, is bound to be somewhat artificial. There are large overlaps and interrelations between the categories: Configuration management and process modelling involve methodologies and tools. Many methodologies are tool-based. Process modelling and configuration management may be strongly coupled.

**Lifecycle or process models** The purpose of lifecycle and process models is to lay down a framework for how a development project should proceed, in effect a framework for project management: To identify and measure progress, to enforce standards, use of methodologies, control access etc. The classical lifecycle model is the *waterfall model* which assumes that the project will proceed as an orderly sequence of phases, e.g. requirements analysis, design, coding, testing and maintenance. Even though its problems are well understood, this model is still popular and forms a common reference for much of this area. In fact, it is often taken for granted in for instance texts on software project management. Other lifecycle models are proposed to replace or complement it, for instance the *spiral model* [Boe88] or exploratory programming ("prototyping").

**Process modelling** is a newer approach, which works from the assumption that no single lifecycle model may be appropriate. Instead, the lifecycle model should be made explicit (process modelling) and used to guide work in an integrated software engineering environment. Process modelling is in its infancy, and there are several competing approaches:

- *Process programming* [TBC+89, OR86] views the progress of the project as the execution of a *process program* written down in an imperative programming language (with concurrency).

- In the *contractual approach* [Dow87] the main organizing principle is work breakdown, in the form of contracts which can be broken down into subcontracts and handed on to other agents.

- Rule-based reasoning, i.e. tasks with pre- and postconditions [KF87, B+89].

- Task networks with dynamic triggering [DGS89, HM88, BE87].

Hybrids of two or more of the above are also contemplated, for instance in the EPOS project (see section 1.2).
Methodologies  *Analysis and design methods* abound: JSP [Jac75], SA/SADT [Ros77], ISAC [L+], HIPO [Sta76], data flow diagrams [GS79], SDL [CCI85], VDM [Jon80] …., just to mention some of the most popular ones.

A taxonomy of analysis and design methods according to Buckle [Buc86]:

**Analysis:**

- Diagrammatic methods
- Formal methods
- Computer-based systems
- Prototyping

**Design:**

- Functional decomposition or stepwise refinement
- Input-output structure
- Structured or data flow design
- Data abstraction
- State-transition
- Program design languages
- Data design

Classical approaches tend to be strongly coupled to the waterfall lifecycle model, which prescribes that analysis, design and implementation follow each other as well-defined phases. Some newer approaches do to some extent do away with this strict phase-separation, e.g. prototyping and “4th generation languages” in the business area, or more experimental approaches like transformational programming [P+83, REF].

Many of these approaches are geared towards data processing approaches, and are not particularly well-suited in other areas. Data abstraction is one of the most generally applicable design approaches, and was a major guiding principle for programming language development during late 70’s and early 80’s. One particular outgrowth has been *object-oriented design* and programming, which is rapidly gaining recognition.

**Tooling and tool integration**  Many of the methodologies above are intended to be used with support tools. In addition we have all the classical tools, like text editors, compilers, linkers, file comparators; or tools for project management, configuration management etc. Tools need to be collected in a framework that ensures easy exchange of data between tools, presents a uniform user interface, enforces
standards and so on. Such support systems (infrastructure) is commonly denoted programming environments (PE) or software engineering environments (SEE), and has been an active area of research for several years.

The UNIX system [Mit81] is an early example of a system where this “environment” thinking has been central: The operating system supplies the infrastructure for integrating cooperating tools. A simple, uniform data representation (“flat” ASCII character streams) plus an extensive set of support tools for that, and operating system features like pipes and input/output-redirection makes it fairly easy to exchange data between tools. But even though UNIX has been very successful, it has also served to highlight the problems and limitations of an essentially ad-hoc approach to integration. IDL [Lam83] is an example of the same tool interconnection paradigm, but with somewhat more structure than UNIX pipes, in that the data structures exchanged between tools are typed.

Currently, most research interest is focused into environments built around a database, where the database is the main integration vehicle. This approach has potential major advantages over integration through simple text streams: explicit and centralized control over data format and semantics, independence of data representation, data integrity, multiuser concurrency and access control. However, the overwhelming disadvantage is performance: It is not possible to represent fine-grained data using current database technology with anything remotely close to acceptable performance.

Configuration management The purpose of configuration management (CM) is to control and maintain an evolving product, with both planned and unanticipated variability. In mechanical and electrical engineering, CM is well established with largely agreed-upon terminology and procedures. Classical approaches to software configuration management (SCM) [BHS80] are adaptations of these established CM procedures. CM is basically seen as a branch of project management, and does often take the concrete form of standardized procedures described in voluminous manuals. The important objectives seem to be to constrain the development process in a way to make it manageable, and to extract the information required for managerial control.

Key activities in classical SCM:

- **identification** and naming of components of software system
- **control**: freezing of baseline configurations
- **auditing**
- **status accounting**

The other, more recent, approach to SCM stems from the SEE and programming languages research, and is much more tool and developer oriented. Important issues are: recording of relevant information, tool support for selecting consistent sets of component versions, regulating and supporting concurrent and cooperating developers. SCM is seen more as an infrastructure issue, not only a high-level, managerial
task. Often, one envisages SCM, or at least basic support for SCM, such as versioning, built into the underlying database or “product archive” infrastructure of an SEE.

A largely open question is whether there is something fundamentally different between SCM and traditional configuration management for “hard” physical items. A software product is its own description and can be directly put under CM, while for physical objects, only the description of the product—not the product itself—is subject to CM.

### 1.2 The EPOS Project

The EPOS\(^1\) project aims at producing a generic, kernel software engineering environment. The project has been running since 1986 at the Division of Computer Systems and Telematics, Norwegian Institute of Technology and ELAB-RUNIT, SINTEF group.

Key technical points are:

- A new versioning model, the *change-oriented model*.
- Version management integrated in database support.
- Strong integration of process and configuration management. (“PM of CM and CM of PM”)
- Process management based on uniform task model with pre- and postconditions, describing tools and human actors.

The versioning model and basic database support is the topic of this thesis, and will be described later in greater detail. To set the context for this work, the components of the project are described briefly here.

**Versioning** We use the *change-oriented* versioning model [Ha88, DLC\(^+\)89, LCK\(^+\)89]. In this model, system versions are specified by setting a list of *options*, identifying the functional properties (changes) to be included in the resulting version.

Each change job modifying the product is associated with a particular set of combination of options, which forms a description of the purpose or goal of the change job, and thus couples versioning to process management.

**Database support** The center of an EPOS environment is formed by the database. The EPOS database supports a data model which is a variant of the Entity-Relationship model, enhanced with some object-oriented mechanisms. The database is uniformly versioned according to the change-oriented model.

\(^1\)EPOS means “Eksperstystem for Program- Og Systemutvikling” (English: “Expert System for Program and System Development”).
To cater for the needs of ordinary tools, components may be checked out of the database and represented as ordinary files in workspaces in the UNIX file system.

**Activity manager** The activity manager oversees and maintains a network of tasks and task types (possibly project specific). Tasks span the spectrum from simple tool activation through tasks modelling human actors to the project itself. The tasks are chained horizontally according to input/output-relationships, and may be nested vertically. The activity manager schedules task activations according to these dependencies.

**Planning** Task types are associated with pre- and postconditions, describing the semantics of the task. To build the task network, the activity manager calls on the planner, which utilizes this information to instantiate and connect tasks.

The planner and the activity manager are actually fairly tightly intertwined. To cater for unexpected results (e.g. compilation errors) planning and execution are interleaved, with replanning after failures.
Chapter 2

Engineering Databases

Until now, database systems have mostly been used for data processing applications, but there is growing appetite for similar technology in other application areas. To see why, let us first consider what the main characteristics of database systems are:

- Efficient handling of large volumes of data.
- Reliable, persistent store.
- Structured, self-describing data storage, making knowledge of data structure and constraints explicit and localized.
- Isolation of application from physical representation and sometimes conceptual organization of data.
- Safe concurrent access to data.

We are in particular interested in databases for support environments for engineering projects. Our main point of entry to the area is through software engineering, but environments for CAD in mechanical engineering, electrical engineering etc. certainly share many of the same characteristics. The demand for suitable database technology has intensified in parallel with the quest for more highly integrated environments. In fact, the database characteristics listed above define what we could call data integration, and are central to any integration effort in this application area.

Unfortunately, commercial database management systems (DBMS) do not form a particularly suitable integration base for engineering support environments. Database systems for data processing applications are designed to work in an environment with the following characteristics:

- Moderate complexity of the structure of the data.
- Very large amounts of rather homogeneous data.
- Relatively constant schema (type definitions).
• Many simultaneous transactions, but each is short-lived and modifies a small fraction of the database.

• Small and relatively closed set of application programs operating on the database.

• Database, DBMS and applications centralized on a single, large computer.

In engineering databases, we meet other requirements. Some of them are complementary to those in the previous list:

• Data have rich, complex structure with a large number of semantical constraints.

• Type definitions tend to evolve over time.

• The number of transactions is fairly small, but each is long-lived and often modifies significant parts of the database.

• The database is operated on with a large and open-ended set of tools.

• Database, DBMS, and applications are distributed over a number of computers.

Although some of the requirements belong to both groups, we note that we can classify the requirements into two groups: The first contains requirements to the data structuring facilities, or data model of the database system. The last group of requirements do largely concern operational characteristics of the database system, but of course the data model will also constrain and affect those.

In the remainder of this chapter, these requirements will be analyzed in more detail. In chapter 4 a few database systems be presented presented as examples of designs made with some of these requirements in mind. In fact, these requirements are not at all unique to engineering databases. Many other “advanced” applications have similar requirements for persistent storage of data, and throughout the chapter, we are mainly using “engineering databases” as a convenient shorthand. However, we are not in the position to make any claims to comprehensive coverage of the requirements for advanced database systems in general.

2.1 Requirements to the Data Model

A data model is a conceptual framework for describing data structures. Since the term “data model” is often misunderstood and misused, we will start with a definition. (See also [TL82, page 10] or [Ull88, page 32].)

A data model is:

• A set of rules for how data types can be constructed (type constructors).
• A set of semantical constraints inherent in the type constructors, or which can be explicitly expressed.

• A set of operations which can be applied to the data.

A schema is a collection of type definitions.

In other words, if we compare a database schema to a program, and a particular database with that schema to a particular execution of the program, the data model corresponds to the programming language.1

Considering a particular application domain, it is important that the constructs in the data model correspond closely to the concepts in the application domain. If the schema designer is forced to create a number of artificial representation types to represent each real world object category, the schema becomes more complex than necessary. Not only is that a nuisance from the more conceptual point of view, but performance will also suffer. As an example, consider a database system supporting "complex objects", i.e. nested object structures, representing them by direct, physical clustering of nested objects. If represented in a relational database, the same structure would "fall apart" into a large number of individual tuples in different relations. In this case, the clustering inherent in the data at the conceptual level is lost at the implementational level, due to a unsuitable data model.

2.1.1 Classical data models

The three classical data models extensively used in commercial database systems are (1) the hierarchical model, (2) the network (or CODASYL) model, and (3) the relational model. See [Dat86], [TL82] or [Ull88] for further reading. Of these models, the relational model is currently by far the most popular, mainly due to its simplicity and mathematical tractability. However, there seems to be a growing consensus that these data models are not particularly well suited for engineering databases—both among workers applying database technology to this area, and among database researchers themselves.

To a certain extent, the reason for this is that the classical data models are not rich enough. Their constructs are distant from the concepts in the application domains we are considering. In the next section, we will discuss features which are useful in the context of engineering databases, but are missing in the relational model and other classical data models.

We will mostly find that these features can be implemented on top of any other data model. In a way, this is analogous to the fact that any function which can be implemented in one Turing-complete language can be implemented in any of them. But being implementable does not mean that an efficient implementation is possible—neither for a particular function in a theoretically Turing-complete language, nor for a particular schema in an unsuitable data model.

1A common misconception seems to be that a data model is the same as a database schema.
2.1.2 Data model features

Object identity

A data model exhibits object identity if it support objects with a distinct existence and identity, separate from the values of the attributes. In contrast, in value oriented data models, objects are nothing but the aggregation of their attributes. Ullman [Ull88, page 22] takes the somewhat extreme view that "object oriented data models" are exactly those which have object identity.

The relational model is the most characteristic value oriented data model. Relation tuples have no identity, but are simply the concatenation of the attribute values. All semantic connections are made through value-based association. In a pure relational model, these cannot even be explicitly stated, although some systems allow "foreign key" constraints to be specified. Even if that is the case, a key is still per definition an attribute which can be used for associative access, and is not the same as an object identity. This does not imply that it is not possible to implement object identity on top of a relational database—i.e. by having attributes which, by convention, are unique and read-only.

At the opposite end of the spectrum, we find binary data models, e.g. SBDM [Abr74]. In these models, an object is nothing but its identity, and all properties of an object are attached via relationships or links. Even if the object does not have any properties (except a type), it still does exist as a distinct object.

An object identity is not simply a pointer or address: First and foremost, the object identity is a conceptual device, while a pointer is an implementation technique. A pointer is normally considered to refer to a specific storage location, as opposed to what is stored in the location, while an object identity is location-independent. Furthermore, an object identity is not necessarily a runtime value that the application can manipulate.

The two main reasons to require object identity are:

- Conceptual: For at least some categories of "real world" objects, it is natural to consider them to have an identity separate from any of their attributes. If a person changes the name, we would still consider him or her to be the same person. Conversely, even if the social security number is not supposed to change, it would be ridiculous to suggest that the SSN is the "identity" (rather than an attribute) of the person.

- Efficiency: Frequently, the total flexibility of arbitrary value-based associations, as in the relational model, is not called for. Without object identities, it is difficult to imagine direct links between objects. To stress the point again: This is not a request for pointers. Object identity is basically what makes mechanisms such as relationships, links, CODASYL sets conceivable.
Complex objects

In the relational data model data structures are “flat”, consisting of homogeneous records. The same is true, to a certain extent, in the network model. On the other hand, many objects we intend to represent in engineering databases have a nested structure, each complex (or composite) object being composed of or containing several subobjects. (For the third of the classical data models, the hierarchical model, complex objects is actually the main data structuring feature.)

There are a number of different approaches to complex objects:

- By aggregation:
  - With aggregate type constructors (record, set, sequence), similar to those found in programming languages. [SS77, KBC+87, LV89]
  - With implicit subobject-relationships (objects may be inserted and removed dynamically as subobjects of a complex object). [DrGhLm87]

- Non-first-normal-form relations, an extension to the relational model where attributes are allowed to have sets, tuples or nested relations as values. [KLW87]

It is possible to implement complex objects in data models where the concept itself is missing: The containment can be modelled through explicit relationships. However, some aspects (variadic aggregates, i.e. sets or sequences, or non-first-normal-form relations, and even relationships themselves) can be difficult or awkward to implement in some models. This is the case in for instance the relational model, where this may involve creating artificial auxiliary relations, and inventing arbitrary key values to link the main and the auxiliary relations.

Some data models incorporating complex objects do offer additional modelling power through the notion of dependent and independent subobjects. An instance of a dependent object type can only exist as a subobject. In the Entity-Relationship model [Che76] this constraint is called an existence dependency. The CODASYL set retention (and set insertion) options offer similar modelling power. Fixed and mandatory set retention means that a record at all times has to be member of a particular CODASYL set, and corresponds to having dependent subobjects.

Another distinction is whether the data model allows shared subobjects, i.e. whether the same object may be a subobject of more than one complex object at the same time.

The Damokles database system [DrGhLm87] allows relationships as well as objects to be subobjects in complex objects. This is a very useful feature which can be difficult to model in other data models.

Performance is probably the greatest benefit of complex objects. Most complex object models encourage physical clustering of subobjects, so an object with all its subobjects can be accessed in relatively few disk accesses. To a certain extent, this mirrors the current state of the art in program development and CAD environments,
where the file system is used for persistent storage: A file is essentially a complex object, even though the price for efficiency is that its internal structure is left opaque to the storage system.

Also, complex objects open the possibility of locking on a variable level of granularity. Very fine-grained locking is expensive, since there is a large overhead for checking and setting locks for practically every database operation. On the other hand, coarse-grained locking (file, table or subdatabase granularity) may also be expensive, because unnecessary locking reduces concurrency and increases the number of update conflicts. In a model with complex objects, it is natural to let a lock on an object implicitly extend to all its subobjects, and accesses to subobjects can then be performed without any further locking. Still, it is possible to set locks on a finer grain if that is all that is needed. So-called granularity locking or intent locking is a well-known mechanism ([Gra78] and [Dat83, section 3.11]), used for instance in the hierarchical IMS database, and this mechanism is more or less directly applicable to complex object locking. Shared subobjects do create some problems in connection with this mechanism, however.

Subtyping and inheritance

*Generalization*\(^2\) is the process of creating a new type as the union over a set of old types, focusing on the similarities and abstracting away the differences. The inverse of generalization is specialization, i.e. creating new types as specialization of old ones. If type A is a specialization of type B (or B a generalization of A) instances of A are normally considered to be instances of B as well, such that an A instance can occur wherever a B instance can. In that case A is said to be a *subtype* of B. The virtue of subtyping is that it allows more generic operations, working at the appropriate abstraction level. It is possible to operate on a “car” object without actually knowing (or caring) whether it is a “truck” or a “van”.

Inheritance is a separate issue, but strongly related to subtyping. Basically, inheritance is a mechanism to avoid redundancy in specifications, not so much an abstraction mechanism. Most commonly, inheritance is at the *type level*, namely that operations and properties are inherited from supertype to subtype. In that case, inheritance is inseparably bound to subtyping.

However, *instance-based inheritance* is found in some, rather extremely behaviorally object-oriented data models. These data models hardly have any concept of data type at all, operations are directly attached to instances, and an object is simply the sum of its operations. Inheritance is along inter-object links. The first object created of a given “type” frequently acts as the prototype object for the others, i.e. defines their methods.

The classical data models do not have any subtyping or inheritance mechanisms at all. It is possible to implement subtyping on top of those data models in fairly clean ways, (namely by linked subobjects for each of the type and supertypes of

\(^2\)Strictly speaking, *type-type* generalization. Token-type (instance-type) generalization is normally called *classification.*
each object), but these implementations pay a high price in performance. There are several extensions to the relational model which offer subtyping as a built-in construct, notably RM/T [Cod79] and POSTGRES [RS87].

User-defined attribute domains

Most data models offer a fairly small and fixed set of domains for the attributes of objects, typically integers and floating point numbers in various precisions, strings and dates. Attribute domains and object types are normally distinct and cannot be mixed. To some extent, this reflects implementation restrictions, namely which data types can conveniently be represented directly as a record field.

A commonly found requirement for engineering databases is that they should offer user-defined attribute domains, such as in programming languages. The requirement is not necessarily that arbitrary domains should be definable, but rather that the pre-defined set of attribute domains in current database systems are not sufficient. For each application area, an additional few predefined and application specific domains may be all that is really needed.

Many applications require long fields, i.e. variable length unstructured data, essentially files. Common usages are for texts, source files, binary object code, bitmap images etc. Operations on long fields are essentially the same as for files. Long fields could of course be represented as ordinary files, with the “long field” attribute itself merely being a file name. But these files are outside the control of the database system, and hence not covered by the normal transaction facilities (concurrency control, logging, rollback and recovery).

Some lists of requirements calls for various exotic attribute types to represent 2D, 3D drawings, bitmaps etc. If such data really are to be represented as attributes, i.e. with no structure apparent to the DBMS, in our view, long fields are sufficient. On the other hand, it might be more appropriate to represent these in a structured fashion, and then these requirements are really requirement to complex object and other data structuring facilities.

As noted above, it is common to consider domains and object types to be conceptually distinct: attribute values are not objects, and objects cannot be attribute values. The reason for this dichotomy is not only ease of implementation, but also conceptual: Most data models have a separate construction to represent relationships between objects, and if objects are allowed as attribute values, yet another mechanism to relate objects is introduced. The exceptions are data models where pointers (or some abstraction of pointers) are the main means of linking objects, e.g. most object-oriented data models, and data models where subobjects are essentially object-valued attributes. (The latter frequently forbid recursive object structures.)

Taking these restrictions, it is not difficult to offer type constructors for attribute domains similar to the type constructors in programming languages. For instance, the Damokles data model [DrGhLm87] offers all type constructors found in the C language, except for pointers: Arrays, enumerations, structures (records) and unions, as well as the builtin scalar types, subranges of scalar types, strings (constrained
to match some regular expression) and long fields. (Note that structures are not considered to be objects.)

**Modelling of behavior**

As defined in the beginning of this section, a data model also incorporates a set of operations which can be performed on the data. Most data models, in particular the classical ones, only have predefined generic operations, and it is not possible to specify new or type-specific operations.

In environments where a limited number of tightly controlled applications are granted access to the database, this might not be a problem: Any common type-specific operations can simply be implemented in the applications (e.g. in the form of a library), as opposed to within the database system itself. As soon as users are allowed to add new applications accessing the database, or interactive database browsing and update tools are used, this becomes unmanageable.

There are several reasons for that:

- **Consistency and security**: Ad hoc updates of the database by generic operations may fail to observe semantic constraints.

- **Redundancy**: The same operation may end up implemented several times in different applications.

- **Database evolution**: Changing the database schema may involve reimplementation of some operations. This is a managerial headache as long as that implementation is not part of the database system and under the control of the database administrator.

A traditional way to address this, especially the first point, is to have the database system checking inherent and possibly explicitly specified *constraints*, rejecting updates leading to an inconsistent database state. In practice, constraint checking is fairly limited, because only constraints of a rather local nature (e.g. uniqueness of keys, cardinality of relationships etc.) can be checked for with reasonable efficiency.

Another possibility is to use *triggers*, which also cater for the two other points to some degree: Triggers fire when specified, discrete events occur in the database, for instance, update of an attribute. If a given condition holds, a procedure is executed (in the context of the database system, not the application). The trigger procedure can test whether an update is semantically valid and reject it if not, or it could perform any additional operations required to return the database to a valid state, for instance inserting or deleting reverse links, deleting dependent objects after delete operations etc. Constraints are basically declarative, while triggers are procedural!

Common to both approaches is that the application or user is accessing and modifying the database by the generic operations, and that consistency is maintained by rejecting operations which would violate the constraints.
An alternative approach is to move much of the type-specific operations out of the applications and into the database, defined in the database schema. In this case, it is the implementor of these operations who is charged with maintaining consistency. The potentially dangerous generic operations can be avoided. This is the main characteristic of behaviorally object-oriented systems, namely that operations are encapsulated with data, and no other access by generic operations is permitted.

Furthermore, if high-level operations are implemented in the database system, communication overhead may be reduced. Much of the overhead of using a database system actually originates from crossing the protection barrier between the application and the DBMS, since they normally run in separate processes. In the case of physically distributed systems, the latency of network communication is an additional severe cost. Any strategy which reduces the number of calls between the application and DBMS is a performance win.

It is not immediately obvious, but there is another potential performance benefit of implementing higher-level operations in the database system: More efficient concurrency control can be gained by using operation-specific locking strategies. In section 2.2.1 an example is presented where there is an atomic increment operation in a database system. If an “increment” lock type is introduced, more concurrency is permitted without update conflicts than if the generic exclusive write lock should be used. Even if these operations cannot run as atomic units, from the view of the database system, they may be aborted and restarted without involving the application.

2.1.3 Object-orientation

Object-oriented data models can be classified in two groups, with structural or behavioral object-orientation [Dit86, Dit88]. A fully object-oriented data model is both structurally and behaviorally object-oriented. Structurally object-oriented data models have object identity and complex objects, which were discussed in section 2.1.2. Behaviorally object-oriented data models have operations, possibly user-defined, which are strictly associated with data objects (or their type) as in object-oriented languages, e.g. Smalltalk[GR83]. It is also common to have subtyping and inheritance in these models, see section 2.1. It is interesting to note that also behaviorally object-oriented models seem to require object identity. The reason is that it is difficult to associate operations with “something” without conferring it with an identity.

2.1.4 Schema evolution

Traditionally, the database schemas have been considered to be stable and unchanging. Changing schemas (at least at the conceptual level, if not so much at the internal level) is a major upheaval, requiring reloading of the database and recoding of many applications. Moreover, the conversion of the database contents itself may not be possible to perform automatically, based on the differences of the schemas, but do often require manually written programs. Some help has been given by external
view mechanisms, but since it is usually impossible to update the database through
views, current view mechanisms are not fully satisfactory in this respect.

Advanced databases, and engineering databases in particular, put greater demands
on the database with regard to schema evolution. For instance, software engineering
environments (and, we suspect, other project support environment) put a premium
on configurability and flexibility, to cater for the needs of particular projects or
individual users. The processes these environments support also have a much more
prominent element of experimentation and exploration than, say, the average data
processing task.

Some work has been done on schema evolution in the context of object-oriented
databases. Basically, there are two approaches:

- Change the schema and convert the database accordingly [BKKK87].

- Version the schema and record for each instance which type version it corre-
sponds to [SZ86]. Operations are reimplemented, if possible, so that the same
operations are applicable to all versions of a type, or exceptions are raised
if there are no applicable operations. (This approach depends critically on a
behaviorally object-oriented system, where all accesses to objects are through
messages and method calls.)

Both approaches have their virtues. Converting the database to a new schema may
take significant time, but after that, the overhead is low. Doing conversions each
time an object is accessed (which is essentially what the second approach amounts
to) spreads the cost over all accesses to the objects and makes normal operation
slower. On the other hand, the second approach allows old applications to continue
to run unchanged, but still access objects created through the new schema.

An approach intermediate between those two is to convert the database lazily, i.e.
convert objects from old type versions to the current as they are accessed. The
high one-time cost of database reorganization is avoided, and amortized over some
interval of time.

2.2 Concurrency Control

The differences between commercial database systems for EDP and engineering database
systems are perhaps largest in the approaches to concurrency control. Traditional database applications have little use for versioning—basically it is the current
state of the database which is interesting. The transaction mechanism is designed
to ensure that concurrent transactions agree about what the current state is at all
points in time. This does not mean that history is not important, but that is usually
handled by creating logs and adding log entries explicitly.

On the other hand, versioning is often essential in engineering databases, both to rep-
resent history (revisions) and parallel variants. In some systems, versioning largely
obviates traditional concurrency control, by making versions immutable. Even in
this is not the case, versioning makes fine-grained concurrency control (with the associated overhead) less important: It permits concurrent access to different versions, not only different objects. Moreover, it is the very notion that there can exist different versions of the same object that permits conflicting updates to be reconciled (merged).

### 2.2.1 Long transactions

In the application areas we consider, the organizational task structure and decomposition of the work to be done must be represented. At the lowest level, tasks are units of work. All access to the database is in the context of some task, and tasks should be coupled to the transaction mechanism.

However, tasks may be arbitrary long-lived. Some tasks involve a single execution of a single tool, and corresponds well to transactions in traditional database systems. At the other extreme, a task could encompass an entire project, and terminate only when maintenance of the product ceases, years after the project was initiated. Clearly, it is not possible to have transactions in traditional database systems mirroring such tasks. For one thing, traditional or short transactions are non-persistent and must normally be initiated and terminated by the same operating system process. But more importantly, the concurrency mechanisms in such transactions are not appropriate for our use. They are designed with rather pessimistic assumptions, mainly to allow update conflicts to be handled automatically. Specifically:

- Serializability is not to be compromised at any time.
- It is reasonable to roll back a transaction.
- There is a small and fixed number of lock types available, corresponding to the generic database operations.

If a conflict is detected (i.e. a deadlock), maintaining serializability implies that the deadlock must be broken, namely by aborting and rolling back some transaction. However, it is not given that this is the only way of resolving the conflict. In particular, since transactions in engineering databases are often controlled by human beings, it might be possible to handle the conflict in a more informed way. Strictly speaking, serializability will be compromised, but serializability per se is not the goal: The goal is to have database updates performed in a consistent manner. Serializability is one (conservative) criterion for integrity. In fact, serializability is a too weak integrity criterion when the agents responsible for two transactions communicate among themselves outside the database system!

The design of a deadlock avoidance strategy trades the overhead of a more effective strategy against the expected amount of work lost to rolled-back transactions. The assumption in a traditional database system is that all the effort spent in executing a transaction is expended by the computer (not some external agent), and that all the history records required to roll back and redo a transaction are available. It is not only in program development or CAD databases the latter assumption does
not hold—for instance it is rather difficult to “roll back” a teller machine that has already handed out the money!

Even though the traditional lock modes (shared and exclusive) are sufficient to ensure serializability, they are by no means necessary! For instance, if there is an atomic operation to increment a data item, there is no need to have mutual exclusion among transactions which only increment the same data item (see [Ul88, page 491]). I.e. by getting an exclusive write lock on the item, we risk delaying other transactions or creating a deadlock even when serializability was not at stake. In a behaviorally object-oriented system, it is conceivable that some operations could be implemented atomically in the database, and have their own lock types and compatibility matrices.

As we mentioned, ordinary transactions are non-persistent. If its controlling agent, e.g. an operating system process, goes away, the transaction has to be aborted if it was not properly committed. In particular, that means that all transactions active at the time of a system crash have to be aborted. The state of progress of the transaction is lost, and there is no way to continue it.

By relaxing some of these assumptions, we arrive at the long transaction concept. By a long transaction, we understand one which is infeasible to roll back if a serializability conflict is detected. This can be because it is simply too much work invested in it, or frequently because it has involved some interaction with an external agent (a human operator, for instance), such that the history needed for a “complete” rollback and redo is not available.

For instance, the programmer cannot be expected to take notes of each coding decision that was made and redo them if the system says “Transaction rolled back, please redo your work since the beginning of the week”. Furthermore, the programmer might work with different transactions in parallel, or have told other programmers to do this or that in order to maintain consistency with the changes done in the failed transaction. In either case, other transactions depend on changes which were not committed and are rolled back. To maintain serializability in spirit, if not in letter, these transactions should also be rolled back, but there is no way for the database system to know that. The situation is entirely analogous to the phenomenon of “cascaded rollback” which occurs when a transaction releases an exclusive (write) lock early, and then itself is aborted after other transactions have read the item the first transaction wrote.

With respect to the list of the “classical” assumptions above, the examples indicates:

- We might achieve serializability in a very naive sense, but because of the hidden inter-transaction dependencies, that does not really ensure consistency in a broader sense.

- We have to find a better way to resolve conflicts among transactions than naively rolling back the most likely-looking candidate.

To summarize, we view long transactions as having the following characteristics:
• Long transactions are persistent, and do not go away when the associated operating system process terminates.

• Serializability in the strict sense is not achievable.

• Long transactions are normally not feasible to roll back.

• Since update conflicts cannot be handled by rollback, conflicting changes have to be arbitrated by some intelligent agent.

There are several reasons why this is feasible in our application area. With respect to persistency:

• The "controlling agent" for a long transaction is a human user, or perhaps some "persistent process" represented in the database system itself.

With respect to update conflicts:

• The "intelligent agent" behind the transactions do exist—namely a human user (designer, programmer) with deep knowledge of the data structures and their semantic constraints.

• The granularity of the concurrency control may be unnecessarily coarse (to reduce the overhead of concurrency control), flagging update conflicts which by deeper semantic analysis are found to be unimportant. For instance, if an exported module interface is locked as a unit, a transaction changing the definition of a procedure will (unnecessarily) be in conflict with another using a different procedure.

• As the "increment lock" example shows, generic lock modes are pessimistic, and will assume conflicts where a locking strategy with lock modes tailored for the specific operations used finds none.

2.2.2 Checkout and checkin

Long transaction mechanisms are commonly implemented in the form of checkout/checkin-operations. Checkout creates a local subdatabase and makes private copies of the objects the transaction is going to operate on, while checkin returns the objects in the subdatabase to the shared database. The actual semantics of checkout and checkin differs from system to system:

• Locking of checked-out objects:
  • "Removed" from the main database and "owned" by the subdatabase.
  • Exclusively locked in the main database and copied into the subdatabase.
  • Protected from writing, but may be read in the main database. (Perhaps most common model, e.g. in Damokles, [ADG+88, section 4.3.9], Gandalf [SKHA86])
• Whether objects can be checked back in to the main database individually (e.g. Damokles), or all checked-out objects are atomically checked back into the main database as part of the termination of a long transaction (e.g. Gandalf).

• Whether there is a single main database for each subdatabase, which all objects are checked out from (Gandalf), or it is possible to check out objects from any database (Damokles).

The next two sections discuss optimistic concurrency control and nested transactions, which both seem to fit well with the checkout/checkin paradigm.

2.2.3 Optimistic concurrency control

Optimistic concurrency control is based on the assumption that update conflicts are relatively infrequent. Thus, it is normally wasted to set and check locks each time a new data item is accessed in a transaction, since there hardly is any contention for the locks at all. Instead of detecting update conflicts as lock conflicts, conflicts are detected at commit time, by examining the timestamps of the various data items touched by the transaction.\(^3\)

Optimistic concurrency control does not, per se, have anything to do with long transactions, and can be made just as strict as traditional, lock-based concurrency control. In fact, strict, optimistic concurrency control will trade a lower per-operation overhead (no locks) against a higher number of transactions rolled back: A transaction which under lock-based concurrency control just would wait some time for a lock, might under optimistic concurrency control proceed and do read or write operations which subsequently causes an update conflict and rollback. However, coupled with long transactions where conflicting transactions are normally reconciled, rather than rolled back, the higher number of conflicts with optimistic concurrency control is not so much an issue.

Optimistic concurrency control requires each transaction to have a local copy of the data it operates on, since the data may possibly be different from what is in the main, shared database. This could be because the transaction itself updated a particular data item, or because some other transaction did so and committed. Neither of these situations is necessarily an update conflict! Compare this with a checkout/checkin-based transaction implementation, where each transaction has a local workspace.

2.2.4 Nested transactions

Facilities for nested transactions are not common in traditional database systems, possibly because there has not been perceived a need for it. In the application areas we consider, transactions represent tasks performed by a number of independent actors, and it is natural to think of a hierarchical breakdown of tasks. Thus, it is

\(^3\)Timestamp-based concurrency control needs not be optimistic.
very convenient to operate within a nested transaction hierarchy mirroring the task breakdown structure.

With a checkout/checkin based system, nested transactions come very naturally, namely by allowing subdatabases and checkout in several levels. Here, we might in fact argue whether a strictly hierarchical structure is too limiting, i.e. whether a transaction should not be allowed to check out objects from more than one transaction or subdatabase.

2.3 Programming Language Interfaces

A database, or perhaps more appropriately, a data model, can be seen as a parameterized abstract datatype (ADT), where the representation of the data is hidden from the applications operating on it. Applications cannot directly get at the persistent data in the database, but have to rely on the operations provided by the database system. Typically, this means that there is a “firewall” between the types and data in the programming language domain and the types and data in the database domain. The data manipulation operators in the programming language cannot be directly used on the persistent data, instead there is a separate data manipulation language (DML).

On one hand, this is a deplorable state, because it means database applications are generally more cumbersome to write. The DML is a separate language, “embedded”\(^4\) in the programming language.

On the other hand, the real reason for this is that the programming language and the database data models are distinct. Programming language data models do not take into account persistence, while classical database data models tend to mirror the data structures convenient for persistent data (records and files). The only satisfactory solution is to define an integrated database programming language where the programming language and database data model is one and the same. No doubt, the reason why such languages have met with very little success is that application developers rightly are extremely reluctant to throw away the existing body of compilers, support tools for and experience with the commonly used languages of today.

Fortunately, the situation is not quite as bleak as this might suggest. Newer languages (e.g. Ada, C++) which are coming into more widespread use have better data abstraction facilities and “syntactic sugaring” mechanisms (for instance overloading of operator symbols). Even if the “impedance mismatch” between programming language and database cannot be completely removed, at least an embedded data manipulation language can be made more palatable.

For a more comprehensive discussion of database interface designs, see [LPS3].

\(^4\)In this section, we use “embedded language” in a more general sense, i.e. covering embedded SQL and similar approaches, as well as data manipulation languages consisting essentially of a set of library procedures (or abstract datatypes, classes, \ldots for programming languages supporting such constructs).
The following sections describe in turn procedure-based and embedded DMLs, DMLs based on extensions to existing languages, and persistent programming languages. This classification is somewhat arbitrary, based on technical points.

2.3.1 Procedure-based data manipulation languages

Practically all database systems have data manipulation languages in this category. The reasons for that are:

- Few commonly used programming languages have extension mechanisms beyond procedural abstraction. This means that any more sophisticated DML cannot be implemented in the standard language, with standard compilers.

- Libraries of separately compiled procedures and libraries is a well-established technology, with support tools existing in practically any development environment.

- The same separately compiled procedures are normally callable from most supported languages on a particular (machine, operating system) platform, giving the DML wide usability.

A common disadvantage of procedure-based DMLs is that they tend to be detail-prone: The procedures have long parameter lists, catering for all possible variants of the call and exceptions that might be raised. The innards of control blocks are exposed to the application programmer. Much general bookkeeping (declaration of buffers etc.) must be done explicitly. The consequences are:

- Verbose application code which is difficult to comprehend.

- Little compile-time error detection is possible.

- Unsafe programming and lack of protection because data private to the DBMS are exposed to the application programmer.

There is a wide spectrum of possible designs for procedure-based DMLs. A number of the most crucial design issues are discussed in the following sections.

Generic vs. schema-specific

It is most common, at least for older database systems, that the database interface is generic and fixed for all possible database schemas. The procedures in the interface are parameterized with respect to the types they are going to operate on. Data are read from and written to the database system in “flat” buffers of some generic, fixed type, where the interpretation of the data is up to the application.

For some database systems, parts of the interface is schema specific and generated by a “schema compiler”. This could be:
• Constants, providing symbolic names for attributes or types.

• Type definitions, providing an interpretation of otherwise “flat” data buffers exchanged with the database.

• Macro definitions, providing type-specific “syntactic sugaring”.

• Type-specific procedures, providing type-safe operations to exchange data with the database, or even schema-specific operations, semantics, constraints etc.

At the extreme end of the spectrum, the entire database interface, as seen from the application, would be generated from the database schema.

In the design, several concerns are weighed against each other: Efficiency, optimization, programmer convenience, reliability, source or object code compatibility after schema evolution. In general, the less generic interfaces combine easier coding and better readability (not so verbose code) with greater opportunity for optimization and, in general, better efficiency. With more generic interfaces, there is a larger degree of interpretation at runtime, but since fewer assumptions on the schema are compiled into the code, there is also greater isolation against schema evolution. This is not always the case: Many, especially older, database interfaces are generic simply in the sense of not being type-safe. In that case most of the burden of ensuring consistency between application and schema is on the programmer, as opposed to a compiler or an interpretative runtime system.

As an example of increasingly type-specific interfaces, consider a function to read the value of an integer attribute \texttt{attr} from an object in the database:

\begin{itemize}
  \item \texttt{intvar = get\_int("attr", \ldots \);} is the most generic function possible. As long as the name of the attribute is unchanged, both source and object code remains compatible, at the cost of a symbol table lookup for each attribute access. The ECLIPSE system [CA87] takes this approach, and the developers comment on how this drastically reduced the number of functions in the interface.
  
  \item \texttt{intvar = get\_int(attr, \ldots \);} \texttt{attr} can be a constant in a schema-specific “include”-file, in which case the code may need recompilation or at least relinking after a schema change. Alternatively, it could be a variable initialized by a call to the database system, in which case recompilation should normally not be required. Efficiency is much improved over the previous alternative, since symbol table lookup can be replaced with indexing.
  
  \item \texttt{intvar = get\_attr(\ldots \);} A schema-specific function is the most efficient alternative, since no interpretation is required at runtime. Some optimization may also be possible, taking into account special access paths or constraints existing for this particular attribute. The effect is analogous to inlining of procedures, where the body is opened up for optimization and there may be opportunities for constant folding.\footnote{In our case, it is of course the schema compiler which performs the optimizations, not the target language compiler.} Unless the function is implemented as a
\end{itemize}
macro or an inline procedure, it would normally suffice to relink the application after a schema change.

**Single-attribute vs. record-oriented**

Some database interfaces offer procedures to set and get the values for individual attributes at a time. In other, entire records, representing all or groups of attributes for a given object, are read or written in each operation. There is normally a small and fixed number of attribute domains (types), and for that reason it is usually easier to design a good, type-safe, interface with generic, **attribute-oriented** access functions. On the other hand, each procedure call to the database transfers less data, so there are potentially a larger number of calls. Each procedure call may be very costly, so the overhead could be significant: Commonly, the database system itself runs in a common server process, requiring interprocess communication for each procedure call.

**One-at-a-time vs. set-oriented**

Most programming languages have very limited facilities for representing aggregates (sets, sequences), while such constructs are prominent in database data models. This means that applications have to access items from database aggregates one by one, according to **currency indicators** maintained by the database system as part of its per-client context. Common operators will be “get first”, “get next”.

However, some database systems allow aggregates to be handled in a somewhat more explicit way, as **iterators** or **cursors**. The representation of the iterator is hidden, and the application program is merely given a **handle** or **token** to it, which can be stored in a variable and passed to procedures in the database interface. Basically, an iterator is an abstract datatype for a set or sequence, and the interface may offer iterator operations corresponding to a large number of the common set or sequence operations (union, intersection, concatenation, selection according to various criteria).

### 2.3.2 Embedded data manipulation languages

Those DMLs which are commonly called "embedded DMLs" are normally built on a procedure-based DML and extend existing programming languages. The extended language is run through a preprocessor which replaces all occurrences of extensions with procedure calls and produces code in the standard host language. After that, the ordinary compiler is used, and the application is linked with the database interface library. The preprocessing is normally done at a very superficial lexical or syntactical level, with no deep semantic analysis, which severely reduces the level of integration it is possible to achieve.

In the latest years, with the rise of relational databases and SQL as a standard data query and manipulation language, **embedded SQL** has become the data manipula-
tion language of choice for many application developers. Embedded SQL lets the
programmer write SQL statements interspersed among normal host language code
(e.g. PL/I, C). The connections between SQL and host language is made through
host language variables which may occur in SQL expressions. Only variables of
simple types, which can be mapped into SQL field types, are permitted.

2.3.3 Language extensions

The main technical difference between a DML based on language extensions and an
embedded DML is in implementation. With an extended language, the compiler it-
self is extended. Consequently, better integration of semantic processing of the DML
and the host language can be achieved, and many nitty-gritty administrative details
can be attended to by the compiler (as opposed to the application programmer).

COBOL is the main example of a language extended with a DML (CODASYL
DBTG [COD71]). Here, the language is extended with:

- DATA DIVISION statements to “invoke” (activate or connect to) database
  schemas.
- Imperative PROCEDURE DIVISION statements to perform actual database
  operations.

In the case of COBOL, the extended language is standardized (but there is also
a standardized, simpler “level 1” language without these facilities). For a more
general-purpose language, there is understandably little or no willingness to add the
great bulk of a DML to language standards. Thus, in practice, DMLs based on
language extensions are not very usable.

2.3.4 Persistent programming languages

Some programming languages are designed explicitly for operations on databases
and incorporate persistent datatypes in their type system. The boundary between
language extensions and persistent programming languages is exceedingly fuzzy.
Many persistent programming languages are modifications or extensions to com-
mon general-purpose languages. We will draw the boundary based on how extensive
the modifications are, in particular with respect to the type system. Essentially, per-
sistent programming languages embodies the unification of programming language
and database data models which was alluded to earlier.

Examples of persistent programming languages are ADAPLEX [Shi81, SFL81] (DA-
PLEX in Ada), ASTRAL [ABR79], Galileo [ACO85], Pascal-R [SM80], PLAIN
[Was79], PS-algo1 [KA87], RIGEL [RS79], Theseus [Sho79].

Since these languages are new languages or major modifications of common lan-
guages, the same problems as with extended languages apply, namely that compilers
and support tools are not generally available.
Some more special-purpose languages may also conveniently be classified as persistent programming languages:

- The so-called "4th generation languages" have strong database integration.
- The SYBASE\textsuperscript{6} relational database [DE88, page 19] has an extended SQL (TRANSACT-SQL) including control flow and procedures.

\textsuperscript{6}SYBASE and TRANSACT-SQL are trademarks of Sybase, Inc.
Chapter 3

Versioning and configuration management

*Versioning* is the capability to store multiple versions of the same object. *Configuration management* (CM) is versioning for complex objects (configurations), where the individual components themselves are versioned. Thus, the main focus of CM is to manage the combinatorial number of potential configurations arising from combinations of the component versions.

3.1 Basic Versioning

In the brief definition of *versioning* above, two terms were used without themselves being defined: "Version" and "same". While the reader will probably have an intuitive understanding of these terms, it is important to keep in mind that the precise definitions depend strongly on the versioning model used. In the following sections we will describe in what ways versioning models differ.

3.1.1 Versioning concepts

Most versioning mechanisms consider object versions to be objects in their own right. A *version group* or *generic object* is the collection of all versions for the "same" object. At an implementation level, the object versions are normally stored by some delta mechanism to conserve space. Object versions are commonly related in some kind of graph structure, a *version graph* representing the “derived-from” relation. We will denote versioning models where versions are explicit objects as “version-oriented”.

Versions, or rather the relationship between versions, can be classified as:

- *Revisions*, representing a development history where each new version supersedes the older ones.

- *Variants*, different versions which co-exist without superseding each other.
Accordingly, there are several possible restrictions which can be imposed on the structure of the version graph:

- No branching, i.e. sequential revisions.
- Tree, with branching representing parallel variants.
- Directed acyclic graph (DAG), where also version merges can be represented.

Change-oriented versioning, presented in chapter 5, is an alternate model for versioning, where the database itself is uniformly versioned. Object versions are not explicit objects, but instead the “incarnations” of the objects in a particular database version.

### 3.1.2 Delta storage

A more technical point in any versioning system is how to store versions in a space- and time-efficient way. Most versioning systems for (source) text, like SCCS and RCS, store multiple versions using some delta storage system, where only differences between versions are stored, not all versions in full.

There are two basic delta mechanisms for text: *Embedded* and *forward/backward* deltas. For embedded deltas the source text is stored in one file, with interspersed information about which parts to include or exclude in a given version. Thus, it is possible to extract a version of the file in one pass over the archive file, and the cost of extracting a version is essentially independent of which version, but increases over time as the number of versions increase.

When forward/backward deltas are used, one version is stored in full, and the differences between this version and the other versions are stored separately. The deltas are commonly represented as editor scripts which modify one version into another. A forward delta converts a version into its successor, whereas a backward delta converts a version into its predecessor. The time to extract a version depends on how many deltas have to be applied. Thus, it pays to store the most frequently accessed version in full and have deltas relative to that one. For instance, RCS stores the last version on the main line of development (trunk) in full, with backward deltas to recreate earlier versions on the trunk and forward deltas on the branches.

For common patterns of use, delta storage saves significant amounts of storage. For instance, for DSEE it is reported that typically 50–100 versions may be stored in the same space that is required for two versions in full. (DSEE also applies some simple compression techniques, such as compressing leading blanks.) The space savings depend more on the quality of the differencing tool used than on the actual delta method.
3.1.3 Applicability of version-oriented versioning

Version groups

For version-oriented versioning models to be applicable, the data model must permit version groups to be formed. If that is to be the case, it must be possible identify versions of the entity\(^1\) to be versioned. It is not meaningful to form version groups unless the entities in the version group can be considered versions of each other, or of the same abstract entity.

First of all, it is not possible to form version groups of something that is atomic, in the sense of being without any internal structure. For instance, in the semantic binary data model, an object is nothing but its identity, and consequently, cannot have distinct versions. Only the links between objects can be subject to versioning. Considering the Entity-Relationship model, it is clearly possible to form version groups of E-R entities. The crucial difference from the binary model is that an entity is really a composition of its attribute values: Again, the target of versioning is really the relationships between the ER entity and the values of its attributes, and the ER entity is versioned only because these relationships are considered part of it.

⇒ Only non-atomic objects can form version groups.

Furthermore, it is impossible to relate two versions as versions of the same abstract entity, or of each other, without some notion of "sameness". This is precisely the essence of the object identity concept. If we again consider the semantic binary data model, its links cannot be versioned, since it is meaningless to speak of a version of a link—without an object identity there is no way to say that one link is a version of another. The same problem is present for ER relationships, which do not have an identity either.

⇒ Only objects with an identity can form version groups.

In fact, we could turn the argument around. If it is meaningful to form version groups of objects, it is probably appropriate to say that the objects have an identity.

Versioning of low-level data models

When we version fairly low-level "bare bones" data models, it frequently turns out that we cannot in a natural way form version groups. The individual data items in those models are primitive, and usually either atomic or without object identity. The semantic binary data model, and why it is difficult to apply version-oriented versioning to that, has already been mentioned. We shall see that there are similar problems for the relational model:

For the relational model, we can clearly (if trivially) identify whole tables in different database versions as versions of each other. In fact, tables are non-atomic and can be said to have an object identity, so the two requirements above are fulfilled. However,

\(^1\)In the generic sense of "something", not the construct in the Entity-Relationship model.
whole tables are more properly compared to types (or more appropriately, classes, if the distinction between intension and extension is important) than to instances.

On the instance level, i.e. for individual tuples, the situation is more problematic. We can consider all tuples in a relation with the same user key to be a version group. However, in a value oriented data model, such as the relational model, the key is just another data value, and could in principle be changed. If the concept version group should have any meaning, the tuple after changing the key would have to be a version of the previous one, but formally it is not, since it does not have the same key. There is no universal criterion to group tuples into version groups! For that we need an absolute object identity, independent from attribute values or relationships for the object, such as we have in the graph oriented data models.

Versioning of relationships

We saw that it is difficult or impossible to version relationships using version-oriented concepts, because the individual relationships (links) do not have an identity. In fact, the case is almost identical to the case for tuples in the relational model. That one relates values and the other relates objects is not important in this case.

There is a deeper conceptual reason why versioning of relationships is causing trouble, which is really a configuration management problem: A consistent configuration of the object versions is just a selection of compatible object versions. Most systems with some versioning of objects have only two kinds of links, with dynamic or fixed version binding. A dynamic link relates version groups and is non-versioned. Thus, it is not possible to express that two objects are related in one configuration and not in another. The only other alternative is fixed links, which relate specific versions. From a configuration management viewpoint, the only reasonable use for fixed links is to constrain configuration selection. What is the meaning of a fixed link to an object version which is not present in the current configuration? And we are still not able to express that specific versions of two objects may be present in several configurations, but are not necessarily related in all of them.

It seems that a reasonable approach for version-oriented versioning of relationships is to embed relationships in a complex object and version that object. In Damokles (see section 4.3.1), this is possible, but the database interface does not support this usage particularly well. (There is no convenient way to tell the system to only consider relationships in a particular complex object version. That is, when navigating relationships, the application code has to explicitly test each relationship found.)

3.1.4 Transparency of versioning

Many existing tools and applications do not handle versioning, but assume they are working on a single, unique version of the database. Also, it is convenient to be able to ignore versioning when designing and implementing applications which are only considering a single version.
At one extreme, versioning can be implemented on top of a non-versioned database, i.e. the representation of versions is explicit in the database schema. Naturally, this means that the application must handle the nitty-gritty details of versioning all the time. Also, unless the data definition language contains suitable abstraction mechanisms, the representation of versioning is duplicated for each type in the schema, leading to vastly more complicated schemas.

If versioning is a part of the data model (and hence of the basic primitives offered by the database system), both these problems may be avoided. Representation of versions is built-in, and the only added complexity in the schema is frequently some indications about what types are going to be versioned and how, for instance, what version graph structures are permitted. The versioning model may also allow a unique version to be bound for each object in the database a priori, before an application is run. In that case it is feasible to hide versioning from the applications.

### 3.2 Configuration Management

As we noted in the introduction to this chapter, configuration management (CM) is in some sense version management (VM) of complex objects—“configurations”. This sweeping generalization hides the fact that the key problems in CM are mostly different from those in VM.

The common theme in CM is to *record* dependencies between components in order to be able to *(re)construct* consistent configurations:

1. Identification and recording of configurations.
2. Freezing baselines.
3. Management of cooperative work.
4. Efficient rebuild of (derived) configurations.
5. Selection of consistent configurations with stated properties.

Classical SCM focuses strongly on the managerial aspects, item 1, 2 and partly 3 in the list above. The other items are stressed more strongly in the newer (partly experimental) approaches to CM, see for instance the proceedings of two recent workshops on SCM [Win88, Tic89]. See also the SCM introduction on page 13.

Although our exposition here is intended to be as general as possible, our primary interest is SCM—i.e. CM of software components, specifically program code (source or object) and to some extent documents. To avoid becoming overly vague, much of the discussion in the rest of the session is biased toward software. Still, it is worth keeping in mind that many of the principles which are discussed are more widely applicable.
3.2.1 Configuration selection

While classical SCM has been mostly satisfied with recording and freezing "interesting" configurations, newer work in the SCM area is also concerned with selection of configurations from intensional descriptions of their properties. The purposes may be:

- Control propagation of new versions into an individual developer’s working configuration.
- Constructing a configuration with a specified combination of optional features.

Similar mechanisms could be used in both cases, but some of the criteria used for selection might be different.

The dominant approach to configuration selection is to use attributes and attribute constraints (for instance Adele [BE86]). Component versions have attached attributes describing their properties, and sometimes constraints on which other component versions (i.e. their attributes) they may be combined with. The overall description of the configuration takes the form of a predicate on the attributes.

Attributes are commonly used to represent functional properties of the components which cannot be automatically or formally inferred. Coming up with an appropriate set of attributes and attribute domains to describe the relevant properties may require deep and intimate knowledge of the software system, and is probably best regarded as part of the architectural design.

The ability to express preferences is a prime concern for such configuration selection mechanisms [LL87]. Instead of just a binary “yes, this configuration satisfies the configuration description” or “no, it does not”, it should be possible to select a configuration which is best according to some specified criterion. If multiple criteria could be specified, there must be some way to rank them in priority, and it should be possible to represent tradeoffs\(^2\). Preferences could take different forms:

- Some function of the selection attributes to be optimized, for instance:
  - last version
  - highest value for the status attribute
  - least space requirement

- Some selection criterion to be fulfilled only if possible (not in conflict with other criteria).

3.2.2 Rebuild

Efficient and automatic rebuild is primarily an optimization problem, to reduce the computer resources and turnaround time used for rebuild. In this sense, the

\(^2\)A tradeoff is in principle a priority ranking of possibly conflicting preferences.
problem is somewhat peripheral to CM. Tools and techniques for performing rebuilds automatically in some optimal fashion are fairly well established: Make [Fel79] and similar tools are in wide use. Make is very generic, tracking only file-granularity dependencies and using timestamps to detect the need for rederivations. But for this precise reason, it is pessimistic in determining when a rederivation is needed.

_Smart recompilation_ [Tic86, GHWN83, EHW89] attacks this problem by utilizing more fine-grained dependency information, i.e. dependencies between parts of files. If the change in an input object was insignificant, i.e. if the object potentially needing rederivation did not depend on any of the changes in the input object, rederivation is avoided. But fine-grained dependencies are by their very nature non-generic and depend on the semantics of the objects involved. Thus, type-specific tools are needed to determine when rederivation is needed. (Labeling this approach “smart recompilation” may be somewhat misleading, because it is applicable to any tool-based derivation where there are fine-grained dependencies at all.)

Another approach to optimize rebuilding is by _caching_ the results of derivations. (This might be seen as more in line with CM in general, i.e. recording the results of earlier builds.) For each derived object in the cache, the bound subconfiguration of input objects used to build it is recorded, along with the (version of) the tool used for the derivation and the switches the tools was invoked with. This information is used as a key in the cache. When a derived object version is requested, the cache is consulted for a matching object before running the derivation. This approach can profitably be combined with smart recompilation techniques: If a precise match for the requested derived object cannot be found in the cache, it may be possible to determine that another object is equivalent by examining the differences between the input object versions.

The build problem is not always strictly a matter of optimization of computer resource usage. In many cases, the elapsed time for a complete rebuild is significant on a project timescale. Large systems may actually require days for a rebuild, and without smart recompilation tools it may not be feasible to keep a consistent derived version of the system at all.

### 3.2.3 Special-purpose CM mechanisms

Many of the problems of maintaining an operating system or tool installation are in principle configuration management problems [Dar89]. An operating system, such as UNIX, with a toolbox approach to tool integration, abound with examples of special-case and ad-hoc CM mechanism catering to specific cases.

**Logical names and search paths**

Logical names and search paths are fairly ubiquitous mechanisms in most operating systems and many other tool environments. The principal purpose of such mechanisms is to delay the binding of a configuration, so that the developer of the software does not preempt the decisions of the installer and maintainer—or user—in setting
up their environment. A logical name (i.e. a variable containing a file or directory name) specifies which one of a number of components or subconfigurations go into a larger configuration.

Search paths introduce an element of configuration selection. Typically, they are used to set up incrementally modified configurations. A typical UNIX “path” setting might be:

\[
\text{PATH=}$\text{HOME/bin:/usr/local/bin:/usr/bin:/bin}
\]

The directories \text{/usr/bin} and \text{/bin} contain the configuration of executable programs as shipped from the vendor. \text{/bin} is the configuration of tools needed at boot time (before the file systems containing the other programs are mounted), and \text{/usr/bin} is needed to complete the configuration of standard tools. The standard tool configuration is incrementally modified by \text{/usr/local/bin}, the site-specific tool configuration, and \text{$\text{HOME/bin}$}, the user’s private programs.

This example highlights one of the problems with such mechanisms. There may be configuration constraints, but these cannot be expressed in a convenient way. For instance, with the path setting above, it is necessary to set up the \text{MANPATH} variable correctly, in order to get the versions of the online documentation pages that corresponds to the program versions selected by \text{PATH}.

A general configuration management system which does some modelling of tool behavior will have to handle these mechanisms in one way or the other. The best way might be to utilize these mechanisms in the way they conventionally are, but doing so, one quickly ends up in a quagmire of ad-hoc mechanisms which do not fit conceptually with the CM tool itself. Thus, it seems better to realize that the functionality the ad-hoc mechanisms provide to a large degree is subsumed by the general CM tool and avoid them as far as possible.

Controlled replication

In a distributed workstation environment it is frequently necessary to replicate software among the machines, to increase reliability and to reduce the load on particular machines. The BSD UNIX Rdist program [UPM, section rdist(1)] controls replication of files over a number of different machines. It is controlled by a script file which is essentially a description of the subconfigurations that exist, what machines they are to be distributed to and in what way they are to be installed on the remote system. For each of the managed configurations, Rdist keeps the copies on the local and remote machines synchronized.

3.2.4 Modularization

In the preceding paragraphs, we have mainly been considering how to describe, record and track \text{existing} inter-component dependencies. Techniques for \text{limiting} the number and complexity of such dependencies are also strongly related to CM.
Programming language features, coding style and other limited technical means may serve to limit dependencies, but most of all a careful system design is crucial. After all, this is what sound design and programming practices are mostly about.

Variability in a software system has to be planned and designed in from the beginning. If not, even the most advanced CM environment will probably not be effective in managing it. Systems with a large number of optional features are frequently designed in a building block fashion. Features are selected by exchanging system components. In such cases, it is necessary to pay special attention to the interfaces to optional components: The potential number of combinations of optional features may be far too large to permit exhaustive a priori testing, and the more dependencies there are across those interfaces, the less the chance is that untested combinations will be usable. In other words, it is important to localize (modularize) specific features to as few components as possible.

Operating systems is an example of a class of systems which generally have a large amount of configurability designed into them. The same operating system should be possible to configure to run on a wide range of computer installations, with different numbers and types of hardware devices, hardware characteristics, workload and tasks. Some operating systems have what amounts to special purpose CM systems to configure and build customized variants, for instance the BSD Unix “config” program [UPM, config(8)].

Config takes an input file which contains a description of the hardware the operating system is to run on (what external devices exist, at what addresses, what interrupt vectors etc.) and various other configuration parameters (time zone, size of disk buffer cache and other kernel tables and buffers, whether to enable disk quotas and resource accounting etc.) On some Unix versions derived from BSD (for instance Sun Microsystems' ) it is also possible to exclude or include various subsets of the system calls. Config produces a makefile for Make and C source files with various configuration tables, device driver and system call dispatch tables and other “glue”. These files are placed in a subdirectory along with the required relocatable files from the kernel distribution, ready to run Make to produce a final executable OS kernel.

The key to all this configurability is that the kernel is explicitly designed for it. Some of the configuration does even take place at boot time, for instance the kernel probes for which devices are actually present and enabled.

### 3.2.5 Distributed environments

Software engineering is frequently performed in distributed environments. Not only is it becoming more and more common to perform development on networks of workstations and servers. In large projects development may take place at geographically distributed sites, even at customer sites. Moreover, for some categories of software (e.g. real-time, embedded, telecommunications), cross-development\(^3\) is

---

\(^3\)Cross-development: Software is developed on one system (the host system) to run on another system (the target system), frequently because the target system is special-purpose hardware incapable of running editors, compilers and other development tools.
the rule. Configuration management in distributed environments pose special challenges, originating from the generic problems of distributed systems:

- Communication cost: All parts of a configuration may not be available locally.
- Redundancy: Duplicated copies of the same entity must be kept consistent.

Many of the distribution problems should no doubt be solved with the general techniques developed for distributed operating system and distributed databases. These techniques generally aim to hide the distribution aspects and provide the abstraction of a single, consistent environment.

However, in many of the real-world distributed development situations, the general techniques are inadequate or inapplicable. CM as a methodology provides another way out, in that it is possible to reckon with a certain amount of inconsistency and permits that to be kept under control. In no way is it possible to maintain the abstraction of a single environment, if development is going on at different sites, and the only communication method is long latency, low bandwidth, such as by mailing a magnetic tape. In such cases, CM techniques must be used to permit parallel development with some kind of merging periodically.

With cross-development the main cause of problems is the turnaround time from development to target system, in particular if the target system is not directly connected to the development system, permitting object code to be directly downloaded. To be able to meet schedules, it is frequently necessary to resort to patching of object code directly on the target system, instead of making the corrections in the source code and go through the entire edit-compile-download cycle again. To our knowledge, few if any CM systems support such practice in an adequate way. In the worst case, source and object code for a module may get so thoroughly out of synchronization that is not any more possible to rebuild the object code from source! (There are some unconfirmed horror stories around that indicate that such things do in fact happen.)
Chapter 4

Versioning and Configuration Management Systems

In this chapter, we will describe some selected systems which exhibit versioning or configuration management, in order to illustrate the design principles discussed in the two previous chapters. We cannot completely cover the area. In particular there are many commercial or company proprietary CM systems for which there is not much published material.

Many of the concepts of versioning originate from restricted file-based version and configuration management systems, which we will start with. We will then describe experimental database systems incorporating some of these concepts.

4.1 File-based systems

4.1.1 SCCS, RCS

SCCS [Roc75, Gla78] and RCS [Tic82] are fairly simple tools to manage versioned archives of source code on a file by file basis. Still, these systems or clones of them are widely used, and their versioning models are even more pervasive.

Both are strongly based on a checkout/checkin paradigm. File versions have to be checked out from the archive in order to do any work on them, whether to read, compile or actually modify. The tools performing checkout and checkin encourage, but do not enforce, that the archive files are collected in a separate archive directory. There is a one-to-one correspondence between the archive file and the checked-out workfile, but this relation is implicit in the file names and is not explicitly represented (except possibly in a makefile or another kind of command script extraneous to RCS/SCCS).

For each file, versions are organized in a tree, where each version has exactly one predecessor, but possibly more than one successor. However, it is much more convenient to handle only versions of the main trunk of the version tree, and in practice, branching is seldom used. Both SCCS and RCS have mechanisms for merging ver-
sions, although the history of merges cannot be explicitly represented in the version tree. (It would require the version graph to have a DAG structure, which the version numbering scheme does not support.) Versions are immutable, with the exception that certain nodes in the version graph may be removed, in particular nodes with no successors (tip nodes).

Version identification follows a two-level scheme, with major and minor version numbers. There is no particular significance to this, except that minor version numbers can be omitted when specifying a version, giving the last minor version for the specified major version.

There are no relations between versions of distinct files (i.e. across archive files). Keeping track of compatible versions is up to project or programmer's conventions, for instance using major version numbers. RCS also allows versions to be given symbolic names (a specific version, or last version, last version on branch, last minor version within a major version).

In SCCS, each checkin of a changed workfile creates a \textit{delta}. Each checkout operation, producing a workfile, includes a subset of the deltas recorded in the archive file. Normally, the version to check out is identified by specifying a node in the version graph. The delta associated with that node and all nodes on the path between it and the root. However, there is considerable freedom to include or exclude deltas, but there is of course no guarantee as to how meaningful the resulting version will be. This is, for instance, the mechanism used to merge versions. Deltas are stored in the archive file in a structure resembling conditional compilation, so-called \textit{embedded deltas}. Control lines delimit pieces of text to be included or excluded if specific deltas are selected. SCCS also allows new deltas to be inserted between previous deltas (i.e. not only at the leaves of the version tree).

RCS uses forward and backward deltas, stored as editor scripts. To speed retrieval, the tip (last) version on the trunk is stored in full. There are backward deltas for the predecessor versions back to the root, and forward deltas from each branch point and along the branch. Version merging is a so-called 3-way merge: The differences between two other versions can be merged into a checked-out workfile.

Both systems provide fairly simple multiuser control. Provided the archive files are only operated on with the SCCS or RCS tools, their integrity are protected with low-level locks against simultaneous update (i.e. checkin) or read and update. On a higher level, a user is not allowed to check in a new version in the version graph unless the user already has a lock on the predecessor of the new version. The normal way to get a lock is to check out a workfile with the lock option set. There may be locks on several versions in the version graph, but each user is normally granted only one lock. The lock only blocks other users from creating a successor version on the trunk. It is still possible to create branches when another user has a lock on a version.

There is simple access control with respect to which users may create deltas. If the archive files are not specially protected, these access control mechanisms can be overridden by using ordinary UNIX tools to modify the archive files. To prevent that, the checkin and checkout programs can be run with the UNIX "set userid"
mechanism.

4.1.2 Make

Make [Fel79] is frequently classified as a configuration management tool, but is actually only targeting the subproblem of efficiently rebuilding configurations. As such it has been highly successful, and there is a huge number of Make clones and extensions existing. The main reason for the success of Make is probably that it is highly generic and usable for a wide range of applications.

Make is language independent. It is only considering file level build dependencies, namely the dependencies of a derived object on (all) the objects used to derive it. This notion of dependency is applicable to any kind of derived object, as opposed to language specific dependencies like "import", "include" etc. The need for recompilation is determined by comparing timestamps. Again, this criterion is generally applicable, as opposed to any attempt to determine whether changes are significant, which by its nature is language-dependent.

Having such general criteria, Make is not exceedingly smart about recompilations which could in principle be avoided. However, it is hardly possible to use more discriminating criteria without loosing the generic nature of Make and making it language-specific in one way or the other. Furthermore, the simple criteria used by Make do not need long time to evaluate. The cost of evaluating more discriminating criteria must be compared to the cost of the redundant compilations eliminated by using them.

A further reason for the popularity of Make is that the configuration description, the "makefile", in most cases serves as documentation for the system. For a moderately complex system, a capable Unix programmer should be able to configure and install it with no documentation beside the makefile and the source code. (Provided the makefile is written observing the normal Unix conventions.)

Compared to more general CM tools, Make has no notion of multiple versions beside successive revisions distinguished by their timestamps. In a context with parallel variants, all derived objects have to be removed whenever source files are replaced with variants. By convention, Makefiles are often written such that "making" the target clean removes all derived objects.

4.1.3 Cedar

The Cedar System Modeller [LS83] is the configuration management system of the Cedar [SZH85, Don85] environment. The Cedar environment is a prime exponent of the immutable version paradigm. Files are never updated, new versions are always created. There is no particular structure among the versions, and versions are just identified by their timestamp. A system model is an enumeration of the files constituting a configuration, down to their unique version identifiers. When a new version of a file is created, the Modeller is notified. If the new version is going to be
incorporated in a configuration, the Modeller will create a new system model.

The system models do not incorporate derived, only primary objects. Everything can be rederived from the primaries, if needed. However, derived objects are stored in a cache. The projection table records the unique object resulting from running the deriver (compiler) on a particular source object with a particular set of parameters (imported modules, switches). This table is consulted before each derivation in order to avoid unnecessary rederivations. In addition, each derived object incorporates a 48 bit hash code computed from the unique identifiers of the source object and the parameters it was derived from. If an object is not found in the projection table, the hash code it would have is searched for in the file system.

The system models only supports traditional configuration management, in that they only record configurations that have been produced. There is little support for constructing configurations: Apparently, it is possible to manually select component versions by editing the system models\(^1\). There is no higher-level (intentional) description of the configurations which can be used to pick component versions automatically according to specific criteria.

### 4.1.4 DSEE

Apollo's Domain Software Engineering Environment (DSEE) [LC84] has a versioning model similar to SCCS and RCS, namely with versions organized in a tree structure. In contrast to SCCS and RCS in combination with Make, versioning and configuration management is more deeply integrated in DSEE: There is a version map enumerating the component versions to use for a configuration of the software system. Using the version map, the system automatically uses the correct versions of files etc. Versions of derived objects are automatically managed and stored in a cache along with their derivation history (bound configuration thread, BCT).

Of particular interest to us is that versioning is integrated in the file system. Versions are represented by embedded deltas, and any version of a file can be reconstructed in a single pass through the archive file. The normal file system operations (open, read, etc.) apply to versioned as well as non-versioned files, and the open operation automatically consults the version map. Thus, applications which are built to operate on non-versioned files can operate on the versioned files in DSEE with no changes. However, since this feature requires cooperation from the file system, it also makes DSEE difficult to port to other platforms than the proprietary Apollo AEGIS operating system.

There is a high-level mechanism for configuration selection:

- The *system model* lists the components going into a configuration (but not their versions, unlike Cedar System Modeller).
- The *configuration thread* describes how to select versions for the components.

---

\(^1\) However, one would assume that there is some garbage collection of unused versions, and in that case, component versions not present in any system model might disappear without warning.
4.1.5 SIO

The SIO system [BLIV87] uses a fairly conventional versioning model, but has a strong focus on configuration selection, expressing compatibility constraints and also on integration with process modelling. Configurations are typed, complex objects, described through user-specified schemas [LV89]. Component selection, as well as compatibility constraints, are expressed through a formalism based on relational database queries, but extended with a preference mechanism [LL87].

The task model [BLS9] attaches tasks to the successor relation in the version graph. A task has pre- and postconditions constraining the attributes of its input and output version, and may be broken down into subtasks: A development which in the internal context is broken down into subtasks, with several intermediate versions, is in the external context abstracted as a single successor arc expressing the complex task.

4.1.6 Gandalf

The Gandalf system [Not85, ES85, SKHA86] is a program development environment based on the Aloe structure editor generator. All software components under control of the environment are stored in the Smile database. Smile manages a sequence of revisions of each component, and is based on a checkout/checkin paradigm. Each user of the environment has an experimental database where components can be checked out and modified, while they are reserved in the main database. Until the experimental database is checked in, the other users continue to see the previous versions of these components. After they have been checked in, there are no mechanisms to avoid seeing new versions. Smile does also not contain any mechanism for configuration selection. There is an automatic busy recompilation strategy, such that each time a component is replaced in an experimental database, recompilations are immediately scheduled to bring derived objects back up-to-date.

All these policies are hard-coded into Smile, which itself is a standalone program, not using the Aloe editor generator itself. Current work in Gandalf aim at extending Aloe to allow replacements for Smile to be built as Aloe editors [KB88, MSK89]. Abstract syntax trees according to different Aloe grammars (segments) may be connected using segment nodes in the grammar. Persistent storage and versioning is managed on segment granularity. The transaction model offers nested transactions and is a generalization of the Smile model. Each transaction is a workspace containing modified segment versions. On commit, the modified versions are pushed up to the parent transaction.

4.1.7 Adele

The Adele system [EGK84, BE86] is a configuration management and archiving system for program source code. The components managed by Adele are considered to be "implementations" and "interfaces", and the structure of the configurations is constituted by the dependency relations ("imports", "implements") between inter-
faces and implementations. A version group (family in Adele terminology) contains
the individual versions of an object, as well as various administrative information.
Versions are organized in a two-level scheme much like SCCS and RCS, major versions and minor revisions.

The most innovative mechanism in Adele is the configuration selection, which is
based on attributes and selection predicates. Each interface and implementation
version has an attached set of attributes. Implementations do in addition have a set
of predicates constraining the selections for the modules they depend on. Attributes
and predicates are stored in the so-called manual, which is attached to the version.

A configuration is specified by a selection predicate and selected by traversing the
configuration structure, adding components satisfying the selection predicate and
constraints from components which are already added. Predicates consist of a pattern
matching family names and version, and a part specifying attribute conditions
on the matching versions. AND and OR combinations of predicates is possible.
Predicates come in four kinds:

- For imperative predicates, the attributes occurring in the predicate must be
  present and have values satisfying the conditions.

- Conditional predicates apply only if all attributes occurring in the predicate
  are defined for a particular version.

- Exclusive predicates specifically exclude versions satisfying them.

- Default apply only if not in conflict with other predicates.

Two system-defined attributes state and stateconf are used to signal system detectable side effects. The state values are:

- "Experimental": no side effect is detected.
- "Int.modif": an imported interface has been modified.
- "Obsolete": a more recent revision is available of the object, or transitively
  one of its components.
- "Incomplete": a component is missing.

In additional, the user can set other state values, like “unusable”, “tested”, “validated”, “distributed”, “official” etc.

The Adele successor Nomade\(^3\) [Bel84] introduced an event handling mechanism based
on active relationships, which allows flexible recompilation and change propagation
policies to be defined [BE87]. Events are raised at appropriate points, so that user-defined event handlers can implement any policy ranging from estimating side effects

\(^2\)Earlier versions of Adele did not do any backtracking.

\(^3\)Apparently some features designed as parts of Nomade are planned to be implemented as part of Adele, while Nomade itself was not implemented.
before undertaking a modification, via immediate and delayed recompilation to recompilation on demand.

Adele has recently been completely reimplemented (Adele-2) and is being used in its first large-scale industrial setting, the maintenance of the Bull GECOS operating system.

## 4.2 Object Management Systems

### 4.2.1 PCTE and PACT

The PCTE OMS (Object Management System) [GMT86] is a generalization of the hierarchical UNIX file system, based on the Entity-Relationship data model. Instead of just "links", "files" and "directories", user-defined relationship and object types with attributes may be defined.

PCTE does not in itself contain any versioning mechanism. Versions of objects have to be implemented as distinct objects, related by an explicit successor relationship. There are, however, bits and pieces of mechanisms which may be used to implement versioning on top of PCTE:

- Default mechanisms for link names (with proper naming conventions: select last version).
- Dynamically specified preferences for link type and name.
- Links with the stability property (maintain immutable versions).

PACT VMC (Version Management Common Service) [OBG+89] is an implementation of versioning on top of PCTE. VMC maintains versions of complex objects\(^4\) in a DAG structure.

VMCS does not offer completely transparent version management. A version group is basically just a collection of objects related by "predecessor" links, but with no explicit generic object. The configuration management tools rely on naming conventions to implement generic objects and more dynamic version binding, which means applications have to access the PCTE OMS through particular libraries or with special tools. On the other hand, within a single version of a complex object, all version binding is fixed, so at this level versioning is transparent for a rather trivial reason.

\(^4\)In PACT terms, a complex object is a cluster of objects connected by links with the *composition* property.
4.3 Database systems

4.3.1 Damokles

The Damokles system [DrGhLm87] is based on the Entity-Relationship data model, extended with complex objects and long fields. Complex objects (structured objects in Damokles terminology) may contain both objects (entities) and relationships, and shared subobjects are possible. A complex object is basically a "bag" containing any number of subobjects of the types declared to be the structure of the complex object. Subobjects can be inserted or removed at will.

The database system offers both (ordinary) short and long transactions. A database consists of multiple subdatabases, belonging to groups or individual users. A long transaction is started in the context of a subdatabase, and the owner of the database is allowed to check out objects from other subdatabases into it. Objects may also be transferred (copied, moved) between subdatabases independent of long transactions. Long transactions have application-defined names which are used to attach to them.

There is no particular structure imposed on the long transactions (e.g. no nesting), and any long transaction is allowed to check out from any other database, as well as to check out and check in any object at any time. In particular, this means that a transaction can check an object back in to where it came from and afterwards abort (which will not undo the checkin). Hence, a long transaction is neither atomic nor a unit of recovery. There is a number of different lock modes available for checked-out objects, depending on whether the checked-out copy is to be modified or just read, and whether the write-protection of the original object extends to its relationship context. Before a long transaction may terminate, all checked out objects must either be checked in or "unchecked out".

Ordinary (or short) transactions are units of recovery. Also, short-term violation of cardinality and uniqueness constraints are allowed within a short transaction, but not otherwise. Short transactions may not be nested.

Objects (entities) can have versions. Versions are in principle full-fledged objects, implicitly related to the generic object: Versions may be the target of relationships and can be subobjects in a complex object (i.e. even if the generic object itself is not). Generic objects and their versions have distinct types, so it is not possible to have a relationship type that can be dynamically connected to either a generic object or a specific version. Object versions may also themselves have versions or be complex objects. The structure of versions may be restricted to linear, treelike or DAG. The versioning does not have any particular connection with the transaction mechanisms (either long or short): In order to perform version operations (insert/delete version), the generic object and all its versions have to be present in, for instance checked out to, the same database. It is possible to check out an individual version to operate on it, but a checkout/checkin pair bracketing operations on the checked out object does not in itself create a new version as it does in e.g. SCCS and RCS. Versions are not immutable, but behave as any other object with respect to updates.

Version handling is not transparent to the applications. It is not possible to set
up a "version mask" or "configuration thread" and afterwards operate on a unique, non-versioned configuration. There are really only two alternatives for an application which navigates relationships in the database: Either a relationship leads to a generic object, in which case the application itself has to navigate the version structure (if it needs to find a unique version), or a relationship leads to a particular version, in which case the version binding is fixed and independent of the context the application is running in. As mentioned above, Damokles still has a fairly unique capability in that relationships may be inserted as subobjects. Using this mechanism, it is possible to version relationships in a more flexible way by versioning the complex objects the relationships are embedded in.

4.3.2 Orion

The Orion system [KBC+87] is specifically designed and built to support CAD applications. It supports complex objects by allowing objects as attribute values. Object-valued attributes are of two distinct classes, reference and composite, where the composite attributes define the complex object structure. In the earlier version of the data model, an object could not be a component of more than one complex object, and component objects could not exist on their own. This restriction was found to be too severe [KBG89], and was lifted. Composite attributes now have two additional properties, dependent/independent and shared/exclusive.

- Dependent attributes correspond to the Entity-Relationship existence dependencies. If there is a dependent attribute of some type, an instance of that type cannot exist unless it occurs as such an attribute.

- For shared composite attributes, the same object may occur as a subobject in several objects.

Orion supports schema evolution [BKKK87]. The approach is to define the list of operations permitted on schemas in great detail, along with the semantics of each. Objects in Orion may be versioned. Both the generic object and its versions are full-fledged objects with their own object identifier, and may be the target of relationships. In contrast to Damokles, a relationship of a given type may be connected to either a generic object (version group) or a specific version. If it is connected to a generic object, the binding is said to be dynamic, and either user- or system-defined defaults will be used to bind a specific version at the time the relationship is navigated. The versioning, in particular the distinction between dynamic and fixed links, causes some complexity with respect to the exclusivity property of composite links.

Orion uses granularity locking, extended beyond the basic scheme in [Gra78] and [Dat83, section 3.11] to avoid explicitly locking component objects when a complex object is locked.
Chapter 5

Change-Oriented Versioning

5.1 Introduction

The Change-Oriented model of versioning was introduced by Per Holager [Ha88]. While this model was originally formulated to handle versioning of text files, we have extended it to allow the same concepts to be used in versioning of a structured database.

Change-oriented versioning (COV for short) can be considered dual to the conventional versioning models, which we will call version-oriented versioning (VOV). In VOV, one of the primary concepts is version, while change plays a secondary role as merely a difference between versions. On the other hand, in COV, the concept of functional change is the primary one, and versions are merely identified by a characteristic set of functional changes.

In VOV, the task of configuration management is to identify consistent sets of versions of components that taken together will make up a product. A bound configuration is identified by a list of versions, one for each component in the product. In COV, the task of configuration management is similarly to identify consistent sets of changes, and a bound configuration is identified by a list of functional changes or properties applying to the entire database.

An important emphasis in COV is to assist the developer in combining (merging) independent changes. In traditional systems one usually attempts to associate functional changes with particular modules, so that independent changes are consigned to different modules. Then functional changes may be combined by independently selecting versions of different modules.

Even though this is a useful design principle in itself, there are several potential drawbacks:

- The module structure is fairly rigid once determined, in particular in a large project. Unanticipated variability that does not fit the planned module structure is difficult to accommodate.

- Beside configuration management, there are other considerations guiding the
design of the module structure: Minimizing inter-module dependencies, localizing functional properties, comprehension, convenient separate compilation, work breakdown. Even though these considerations all tend to pull in roughly the same direction, the fewer compromises one has to make, the better.

On the other hand, COV keeps the versioning and configuration management largely orthogonal to the module structure. Functional changes may apply to any number of modules, and the model easily accommodates independent changes applying to the same module(s). Moreover, the model incorporates the bookkeeping necessary to record the interdependence between the changes. Hence, the designer is left more freedom to design module structure according to the other criteria mentioned above.

---

Figure 5.1: Change-oriented vs. version-oriented versioning

---

Figure 5.1 indicates the distinction between version selection in change- and version-oriented models. For change-oriented versioning, the choice selects a version of the entire database, shown as a “slice” in the figure. The individual object version is a section the database version slice cuts through the generic object. The same object version may occur in multiple database versions. For version-oriented models, all object versions occur as individual objects in the database, and a separate version map indicates which versions to use.
5.1.1 Example

Before proceeding, we will give a small example to give the reader some intuitive understanding of change-oriented versioning.

Consider a program product that is delivered for IBM-PC, Sun and VAX machines under the operating systems DOS, Unix and VMS. There is no Unix version for the IBM-PC, and the Sun version may be used both on “dumb” terminals and under the SunView window system.

To describe this product, we introduce functional changes identified by the following options:

- **IBMPC, Sun and Vax** describe which processor type the product is going to run on.
- **DOS, Unix and VMS** describe the target operating system.
- **SunView** tells whether the SunView window system is in use.

In addition, there are a number of auxiliary options that normally are not selected directly.

- **16bit** is set if the program is going to run on a 16 bit machine.
- **UseMouse** is set if support for a point-and-click interface should be included.

To describe the legal combinations of options, we specify a boolean expression, the *validity*. In this case, the validity is the conjunction of the expressions in figure 5.2. We note that at the lowest level the validity is really a boolean expression built with only the boolean connectives $\wedge$, $\vee$ and $\neg$. However, the hypothetical syntax above illustrates how we intend the presentation to the user to look. The terms of the validity are grouped in two sets (indicated with horizontal lines in the table). The upper group represents constraints that reasonably can be considered inherent, while the lower group represents restrictions in the implementation. The assignment of a constraint to either of the groups is not absolute—for instance, the product might later be supported for Unix on PCs also—but as a rule, the inherent constraints will not be changed as a result of routine editing jobs.

Imagine the user wants to build a Sun version with window support. The options **Sun** and **SunView** are chosen. (Actually, it would suffice to choose **SunView**.) With these options given explicitly, the validity implicitly determines the value for all the other options, as the reader can convince him/herself about. On the other hand, if only **Sun** was chosen, the choice would not be complete, since it would not be any way to determine whether to choose **SunView** or **UseMouse**.

As soon as a complete choice is made, the user can start accessing the product database, and will see the version corresponding to the option choice for each component. It is also possible to access the database even if the choice is not complete, i.e. some
MutuallyExclusive(IBMPC, Sun, Vax) Make sure that one and only one of those is selected. (I.e. "enumerated type").

MutuallyExclusive(DOS, Unix, VMS) ditto
DOS ⇔ IBMPC Only DOS on IBMPC, and vice versa.
VMS ⇒ VAX VMS is only running on VAX.
SunView ⇒ Sun ∧ UseMouse If SunView, we must be on a Sun; and in addition include the mouse support.
IBMPC ⇔ 16bit IBM-PC's must be handled as 16 bit processors,
Sun ∨ VAX ⇒ ¬16bit but it is not allowed to choose 16 bit when the processor actually is 32 bits.
IBMPC ∨ VAX ⇒ ¬UseMouse No support for mouse for IBM-PC or VAX.

Figure 5.2: Example validity

of the options are left unspecified. In that case, the user may without further ado access any component which is not affected by the unspecified options. Finally, special tools can handle multiple versions and allow the user to view and edit other components, indicating the effects of the unspecified options. [SBK88] gives an example of such a tool, incidentally in a framework that has much in common with change-oriented versioning.

Assume that we intend to add DecWindows support to the product. To do so, we first add the new option DecWindows describing this change. However, we realize that since DecWindows is based on the X Window System, we can probably reuse a lot of the DecWindows changes if we later decide to make X versions for other machines. Hence, we add an auxiliary option X11.

Initially, neither of those options may be chosen—nothing of them is implemented! We therefore "and" the old validity with ¬DecWindows ∧ ¬X11 as soon as these options are introduced.

First, we want to do as much of the X11 stuff as we need for DecWindows, and start an edit task for that purpose. We specify for which option combinations the task's changes is going to be visible by setting the task's ambition (a boolean expression) to X11. At the end of this task, we still don't have any runnable X11 version, so the validity is left as it was. Then we set out to implement the option DecWindows. We know it is going to be meaningless to choose DecWindows unless we chose X11 and are running on a VAX. Therefore, we decide that we are not going to worry about the combination of DecWindows with any other options, and we use DecWindows ∧ X11 ∧ VAX as the ambition, instead of only DecWindows. Then the editing and debugging
starts. It turns out that we do not manage to implement all of the ambition, only for the Unix operating system. The validity of this edit task then ends up as

\[ \text{DecWindows} \land \text{X11} \land \text{VAX} \land \text{Unix} \land \text{inherent constraints} \]  

(5.1)

When the edit task is finished, the validity is extended by "or-ing" with the task validity, and in that way, we release our changes to the other programmers.

### 5.2 The Change-Oriented Model

The change-oriented model makes heavy use of sets, expressed by their characteristic predicate, a boolean expression. In the discussion below, we talk interchangeably about sets and boolean expressions, glossing over the distinction. The reader should keep in mind that the relevant concepts are most easily understood as sets, even though the representation of these sets is in terms of the characteristic predicate, and the set operations union, intersection and complement are implemented as the boolean operators or, and and not.

The following section recapitulates the change-oriented model of versioning, largely following the original description by Per Holger[Ha88], except that the treatment of the validity concept is somewhat different here. The application of change-oriented versioning to text files, described in the same report, is summarized in section 5.6.2.

#### 5.2.1 Basic versioning

**Logical and physical changes**

In the change-oriented model, the variation in a product is described by a set of options. Each option is a boolean variable, and is associated with a specific change in the external properties of the product (functional change). The value True for an option indicates that the associated functional change or functional property is going to be included in the product. A functional change could be a particular functional feature, a bug fix, a performance improvement or something similar.

Many delta-oriented versioning systems seem to assume a one-to-one correspondence between the functional change and the physical changes needed to implement it. Merging of deltas is simply a cumulative application of physical changes. We believe this to be a major flaw, and the change-oriented model distinguishes strictly between physical and functional. It may well be that functional properties are not conceptually in conflict, even though the physical changes implementing them are. In fact, there may be fairly subtle interactions between different functional properties, such that the same functional change is implemented in very different ways according to what other functional changes it is going to co-exist with! Furthermore, not all components of a product need be affected by a functional change at all.
<table>
<thead>
<tr>
<th><strong>change</strong></th>
<th>Externally visible, functional property of system.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>option</strong></td>
<td>Boolean variable, each option selects whether a particular change is applied or not.</td>
</tr>
<tr>
<td><strong>choice</strong></td>
<td>List of option settings.</td>
</tr>
<tr>
<td><strong>complete choice</strong></td>
<td>A choice where enough options have values that a unique version is identified.</td>
</tr>
<tr>
<td><strong>version</strong></td>
<td>Determinate value for database or individual objects produced by applying (setting) a choice.</td>
</tr>
<tr>
<td><strong>fragment</strong></td>
<td>Piece of database that is versioned as a unit.</td>
</tr>
<tr>
<td><strong>visibility</strong></td>
<td>Boolean expression determining for which choices a fragment is visible.</td>
</tr>
<tr>
<td><strong>edit task</strong></td>
<td>(Transaction) Unit of work, modifying database.</td>
</tr>
<tr>
<td><strong>ambition</strong></td>
<td>Context of edit task: Boolean expression determining which choices the editing changes in this task are made visible for.</td>
</tr>
<tr>
<td><strong>primary ambition</strong></td>
<td>The part of the ambition where no overlap with other tasks is permitted.</td>
</tr>
<tr>
<td><strong>stability</strong></td>
<td>Boolean expression determining choices for which the associated versions cannot be changed.</td>
</tr>
<tr>
<td><strong>validity</strong></td>
<td>Boolean expression determining which choices produce valid or completed versions.</td>
</tr>
</tbody>
</table>

Table 5.1: COV glossary

**Selecting a version**

Each job which is going to access the product database has a context describing which version(s) it wishes to see, out of the total set of versions. This context consists of a set of \((\text{option}, \text{value})\) bindings, called a \textit{choice}, see figure 5.3. A choice need not contain a value binding for all existing options. In that case, it is possible that the choice specifies multiple versions for some components in the database, depending on which values are given to some of the as yet unspecified options.

A \textit{complete choice} is a choice which determines enough options that for the part of the database the job is accessing, a unique version is produced. Note that a choice which fixes a value for all options must be complete, but the converse it not necessarily true: Even though some options remain unspecified, these options do not necessarily—in the context of the rest of the choice—affect the version produced. Also, as the region of interest in the database grows smaller, fewer options are generally needed for a complete choice.

A \textit{version of an object}\(^1\) consists of a collection of fragments with visibilities evaluating to \textbf{True}. We will define the visibility for this version as the intersection (logical and) of all these fragments. This intersection consists of all choices which will result in

\(^1\)An object may be a text file, a relational table, an E-R entity, etc., depending on the data model in question.
an identical version for this object.

**Delta mechanism**

The "database" to be versioned, whether a collection of text files or any other structure, consists of a collection of fragments. For a text file, the fragments could be text lines, or preferably arbitrary chunks of text. For a relational database, relation tuples might be fragments. In a particular version, some of the fragments are visible, others are not. Each fragment has an associated *visibility*, which is a boolean expression. For a given choice, the collection of all fragments where the visibility evaluates to True constitutes the version corresponding to this choice. Figure 5.4 shows the simple case where the fragments are lines of text in a text file. Shaded lines are excluded from the current version because the visibilities are False.

This scheme assumes that there either is no ordering among the fragments (as with relational database tuples), or that the ordering is the same in all versions. The last assumption is barely reasonable for our primary application, program source code, and hardly acceptable at all in general. If a fragment is moved, this has to be handled as if the fragment is deleted and reinserted somewhere else. Thus, the result is that the same fragment occurs twice, in different positions and with disjoint visibilities. The space overhead of duplication is less significant than the fact that the versioning system looses track of the correspondence between versions: If the fragment is subsequently changed, the changes will not be automatically propagated into the other copy of the fragment.

However, it is possible to represent the sequence explicitly. The sequence might be represented by explicit links, where the links have visibility. In a text file, moves can be represented by control commands. A move command encloses two text blocks, and depending on whether the move is "visible", the text blocks are produced in one or the other order.
Stability

A crucial property of configuration management is to be able to retrieve old versions intact, for instance those versions of a software system that were delivered to customers. It is important to understand that versions are not stable or immutable per se in the change-oriented model: Editing does not produce a new version, leaving the old version unchanged, as in many version-oriented systems. Rather, editing modifies a version in-place.

For that reason, the change-oriented model requires a specific mechanism to guarantee immutability for selected versions: The stability is a boolean expression defining which choices are guaranteed to be stable, that is, if the same choice is used to check out a database version now and sometime in the future, the versions will be identical. Only those versions resulting from non-stable choices may be changed. As a consequence, the stability is monotonic: The set of stable combinations may never shrink. Otherwise, a stable version could later become non-stable and might then be changed.

5.2.2 Validity

Not every possible combination of options is valid. In the change-oriented model, the basic way to describe the set of valid combinations is through a boolean expression over options called a validity. Validities can be used for two somewhat distinct purposes:
- Indicate inherent or intended restrictions on combinations of options.
- Indicate correctness or completeness of implementation for versions.

The example in section 5.1.1 is focusing on the first use of validity. Some combinations of options may be inherently meaningless, not because of missing implementation, but because of the nature of the problem. For instance, options may be mutually exclusive, such as the options to express operating system dependency in the example: **DOS, Unix and VMS**. (Essentially, in this case we have an enumerated variable with three possible values, rather than three separate options.) Another common case is when some variability is meaningless unless a particular selection is made somewhere else: Unless the system has a full-screen interface, it is probably meaningless to choose whether it is going to use a mouse or a trackball, for instance. In the following, we will assume that there is one validity describing inherent “combinability” of options, and denote that as the inherent validity.

When used to express correctness, the fundamental change-oriented model does not interpret validity in any way. “Correctness” is relative to the criterion used to determine it. We can imagine a number of different levels of correctness, for instance arranged in a hierarchy:

1. No overlapping changes
2. Syntactical correctness
3. Static semantical correctness
4. Simple module test successful
5. Exhaustive module test successful
6. Integration test successful
7. Acceptance test successful

Which validities there are and the criteria used to determine them, which rules there are to govern the transition among them etc. are all project-specific information. We will simply assume that there exist \( n \) validities, \( V_i, i = 1 \ldots n \).

Validity does not interact with the basic versioning mechanism, and is mainly a mechanism to classify choices or versions for user level tools. In principle there is a total validity associated with the database as a whole, which evolves as the database itself evolves. In practice it may be convenient to split the total validity into sub-expressions and structure these in some fashion: Holager[Ha88] splits the total validity into option validities associated with each option. The total validity is the conjunction of all the option validities. Intuitively, the option validity expresses the contribution to the total validity made by the specific option. A choice or the database version resulting from this choice is said to be valid if the total validity evaluates to True when the option values from the choice are substituted in it.
Note that it is permitted to make a non-valid choice and check out a database version from that. This may seem a little counter-intuitive at first, but this is in fact the way work is done: A non-valid version is checked out, edited and verified. If it passes verification, it is made valid as it is replaced in the archive. (See 5.2.3 and 5.3.)

5.2.3 The editing process

In version-oriented models, an editing task is seen as something that takes versions of one or more input objects, edits them, and inserts new versions in the archive. Thus, introduction of variation in the archive and implementation of the variation is inseparably linked. Moreover, the structure of the version graph is constrained by the structure of the tasks creating them. A common and well-known example is the need to make branches in the version graph to accommodate parallel work, and then merging the resulting versions afterwards.

When using a change-oriented configuration management tool, introduction of new options is decoupled from the edit tasks implementing their functionality. The planned variability is specified by introducing options, as well as inherent validities describing their allowed interaction. An edit task works on a set of choices, replacing the versions corresponding to these choices in-place by new versions, hopefully implementing the functionality these choices describe in a better way than the previous ones.

Updating the database

Edit tasks Updates to the database are normally performed in a long transaction. The user checks out the files or other objects that s/he intends to work on, and checks them back in to the database afterwards. We call such a long transaction an edit task. Updates could also be performed directly in the shared database, and this will be discussed later.

Note that there is no difference between an edit task implementing an option for the first time and an edit task integrating the functionality of different options. The first is also an integration task, integrating the new option with some combination of the previously known options. In contrast, in the version-oriented models, there is a difference between adding new versions at the tips of the version graph and merging two (or more) versions.

Figure 5.5 shows the different sets of choices involved in an edit task. Refer to the following paragraphs for explanations.

Creating an edit task To create an edit task, the user determines an ambition to use for the task. The ambition is the set of choices that the user intends to work on in this task, expressed as we usually do as a boolean expression. At a more abstract level, the ambition is the user's declaration of the purpose of the task, and thus is our link to process management. The effective ambition (i.e. the ambition that is actually going to be used) must not contain any stable choices, since the versions
corresponding to them are frozen: We are later going to use the ambition to update the visibilities of the fragments this task did something to. This check, that no ambition ever overlaps the stability, is sufficient to implement stability.

Unless the user is using a tool that is capable of presenting and manipulating all versions within the ambition, it is also necessary to make a choice to determine one unique versions out of this set to be checked out to the workspace. This choice must of course be within the ambition, i.e. imply the ambition, when both are considered as boolean expressions.

During the task, the user may freely switch the choice from one point to another to verify how the new changes fit, or to make changes that cannot be automatically merged into all versions in the ambition. It is also possible to fork off subtasks with ambitions contained within the ambition of the main task. Thus, it is possible to make changes which are explicitly known to be applicable to only a subset of the versions. A plausible way of working would be to start out with the largest possible ambition, select the “most general” choice within the ambition and implement that choice, paying minimal attention to the way the changes integrate in the other choices. Afterward, the main ambition is partitioned in smaller ambitions to perform
more specific changes. This does continue, possibly recursively, until the whole ambition or those parts of it which was aimed for are implemented.

This breakdown of ambitions into smaller ambitions, implemented in subtasks, is comparable to the breakdown of contracts into subcontracts in Istar [Dow87].

**Committing an edit task** After an edit task is completed, the changes performed are reflected in the main database, by modifying visibilities and possibly inserting new fragments. The effects of the task are also reflected in updates of validities and stability. In effect, parts of or even the whole ambition may be added to the stability or any of the validities. For each validity, the set of choices is determined for which the updates done within this edit task are valid. The corresponding expression is called the *task validity*. The task validity is equal to or a subset of the ambition.

The way to determine the task validity $V_i^T$ depends on the specific criterion associated with this specific validity. It could be determined in a number of ways, from “programmer intuition” to a formal, exhaustive test of all versions covered by the ambition. For text files, Holager presents the heuristic “neighbor criterion” [Ha88], see also section 5.6.2.

After $V_i^T$ is determined, the new total validity $V_i'$ is determined as the union of the old validity $V_i$ and $V_i^T$, namely

$$V_i' = V_i \cup V_i^T$$  \hspace{1cm} (5.2)

To underscore the similarity between validity and stability handling, we will denote stability as $V_i$. The task stability $V_i^T$ identifies the choices to be frozen after the task completes.

**Updating visibilities** Prior to updating the stability and validities, the actual updates performed during the edit task are merged back into the database. Added material is added to the database, with proper visibilities. Deletions and in-place updates are handled by updating the existing visibilities, in detail:

- **Additions**: The new fragment is added to the database with a visibility equal to the ambition:

$$S' = A$$  \hspace{1cm} (5.3)

- **Deletions**: The fragment is not to be visible anywhere within the ambition. The new visibility $S'$ is the set difference of the old visibility $S$ and the the ambition $A$, computed as:

$$S' = S \setminus \neg A$$  \hspace{1cm} (5.4)

- **Update in place**: This is in principle treated as a deletion followed by an addition, but the new fragment is not given the whole ambition as visibility. The old fragment with contents $C$ is split into a fragment with visibility

$$S' = S \setminus \neg A$$  \hspace{1cm} (5.5)
and content $C$, and a fragment with visibility

$$S'' = S \land A$$

(5.6)

and the new content $C'$. The new visibilities are non-intersecting, and their union is precisely the old visibility.

For those cases where the ordering of the fragments is significant (in particular text files), Holager also discusses moves of fragments. While a move of for instance a text block always can be treated as a deletion followed by an insertion, this is not convenient in practice, since the information is lost that this is the same text block, occurring in different versions. Hence, later updates to the text block will have to be duplicated. Changes done in one version do not automatically propagate to the the other versions where this text block occurs.
The update rules above are the most generally applicable ones, not the only possible ones. They have the following properties:

- **Visibility** is always strictly partitioned, i.e. split into two disjoint visibilities whose union is the original visibility.

- **Changes** are made visible to the whole ambition.

Both of these could be relaxed under some circumstances. For instance, Holager uses one specific criterion for validity and thus has a single validity. In this case, it is reasonable to use the (task) validity in the update rules. Specifically, for deletions (including the implicit deletion in update), the task validity $V_T$ is used in place of the ambition $A$. The rationale is: All our changes must be visible at least within the task validity. They should be visible at most within the ambition. In the non-valid part of the ambition—precisely since it is not valid—we have considerable freedom to do what is most convenient for us. In this case, that is to make as many fragments as possible visible. Hence we restrict deletions to the minimum and expand additions maximally. However, we shall later see that this convention is not convenient in all cases, for instance when the change-oriented model is applied to the relational data model, see 5.6.3.

Consider for instance an update: Both the new and the old version of the fragment will be visible to a later edit job, which can choose which one is correct—or replace them both with a third, possibly merging them. If we used arbitrarily either the ambition or the validity to update visibilities, either the new or the old version of the fragment would have been visible. If that was the wrong one, one could of course delete that and re-enter the other; but in that case we would introduce redundancy in the database, since we cannot expect the version management tool to detect that the newly entered fragment actually is equal to one already present in the database.

**Determining the ambition** The above discussion of the ambition glossed over some fine points.

1. The entire ambition must be non-stable, i.e. $A \Rightarrow \neg V_0$.

2. Some choices are inherently non-valid, and should not become valid as the result of an edit task.

3. Still, we intend the ambition to be an intuitive and concise characterization of the choices the user intends to make valid.

Re 1 and 3: We cannot expect the user to specify a non-stable ambition. Consequently, the specified ambition has to be modified into an effective ambition which does not include stable choices. In principle, this can be computed as $A_E = A \land \neg V_0$. However, for purposes of tractability, we would like to keep ambitions in particular simple form, namely as a product of negated and non-negated options. To keep $A_E$ in this form, it will be necessary to remove somewhat more of $A$ than the intersection with $V_0$. 

68
Requirement 2 can most easily be handled by using a separate validity, say $V_1$, to express such dependencies, and constrain the corresponding task validity $V_1^T$ to always be null. Ambitions are permitted to contain inherently non-valid choices, even if these cannot be made valid.

**Adding a new option**

New options can be introduced at any time there is a need to distinguish a new functional property of the system from what has been present before. The new option name is registered, along with any refinement of the inherent validity which is required to describe its interaction with the other options. Normally this will be controlled by some access control mechanism, to avoid uncontrolled evolution of the version structure of the system. The access control is independent of the basic change-oriented model which is discussed here.

![Diagram](https://via.placeholder.com/150)

*Figure 5.7: Adding an option*

At the conceptual level, addition of an option doubles the number of choices by splitting or creating a "mirror image" of each previous choice. A choice and its mirror image starts out with the same associated version, but after editing either of them, the corresponding versions diverge. This can be used to permit diverging lines of development, with continuing development on both lines, as well as freezing of versions, or continued development based on versions from stable choices. For instance, to freeze versions, introduce an option distinguishing the the corresponding choices from the ones where development has continued, and make the frozen choices stable to prohibit further development under them.
After the option $O_{\text{new}}$ has been added, there has not yet been done any work to implement the functionality associated with this option. To express that, old validities can be $V_i$ updated:

$$ V'_i = V_i \land \neg O_{\text{new}} $$  \hspace{1cm} (5.7)

With the new validities $V'_i$, no choice containing the option $O_{\text{new}}$ is valid. The same operation is performed with the stability, to ensure that none of the newly created choices are stable.

The "mirror image" semantics alluded to above turns out to be exceedingly simple to implement, namely by leaving the visibility untouched. In this way, the initial versions seen are independent of the state of $O_{\text{new}}$ in the choice.

**Transaction example**

Figure 5.8 shows a possible scenario for implementing orthogonal functional properties of a product. In the left-hand side of the figure, the choice space is represented. The choices of the transactions are indicated by numbers and arrows next to the corresponding node, and the ambitions by shading.

Option A is first introduced and implemented. Then a new feature of the product is planned, and an option for it, B, is added. This creates two new points in the choice space, representing the product with only feature B, and with feature A and B combined. The base version and the version with only A are not affected. The transaction implementing B does also propagate its changes into the merged A and B version, but does not complete the implementation of that. After this (or possibly in parallel with the B transaction), it is decided that the A version needs to be enhanced. The changes are also propagated into the merged version, but still without completing that. Finally, a transaction focuses on the merged version and completes that, using the material left in it from the previous transactions.

In figure 5.9 it is shown how a new option is introduced, representing a revision of the system. First, the general parts of this revision is implemented. Then the features A and B are updated in parallel, and finally these are merged.

**5.2.4 Configuration descriptions**

The basic level of options, which we have been discussing up to this point, is too detailed and low-level to be very useful for the ultimate user of the database. For instance, we only have binary options, while some features of a system might be more appropriately described by attributes with enumerated types. There are also a few other properties which are outright undesirable:

- There is no way to specify preferences: A choice is either complete or non-complete. There is no way to make a non-complete choice, specifying the required properties, and have the system complete the choice according to explicit or implicit preferences.
- Stability cannot be revoked, i.e. once a choice becomes stable it is impossible to return it to a non-stable status.

- It is only possible to specify simple boolean validities, while a more graduated measure is generally required.

In summary, a more user-oriented layer is needed on top of the boolean options and expressions. The user-oriented version selection mechanism could designed in several ways. We will first describe abstractly what its purpose is and its interface to the rest of the environment. Afterwards a concrete mechanism is proposed. However, the discussion in later sections is not dependent on the details of that mechanism.
Figure 5.9: Implementation of revision option
Configuration description requirements

The user-level specification of a configuration of a system is called a configuration description. A configuration description is evaluated to produce a choice (set of option settings) and a selection of components in the database (a configuration in a more traditional sense).

Before proceeding, we shall make a couple of observations of how this compares to traditional, version-oriented models:

- The choice in COV corresponds to the bound configuration (i.e. list of component versions) in VOV. Recall that options and versions have dual roles in VOV and COV.

- Several version-oriented systems have a more user-oriented configuration selection level, above the mere raw selection of versions for each component. This corresponds to the configuration description level in COV, which we are about to describe.

- Component selection as part of the process of identifying a configuration is really a version-oriented concept. If we were taking a purist’s view of change-oriented versioning, we would exclude component selection and simply consider a configuration to be a version of the entire database. With this definition of configuration, a complete choice would uniquely determine a configuration, and vice versa.

In principle, it would be possible to select components indirectly through the choice (i.e. by setting options). Particular components might not be visible at all for some choices, or, with versioned relationships, not reachable. In practice, it is convenient to have a more direct way of selecting parts of the product, but even though we include this mechanism in the configuration description mechanism, it is largely unrelated to the mechanism for selecting options. Therefore, we will ignore this aspect of configuration descriptions for most of the remainder of this section.

The options are represented in a separate part of the database, and are entities with attributes and relationships on their own. In the most abstract sense, a configuration description is simply considered to be query on this part of the database, returning as its result a list of options (to be given the value True). This part of the database will evolve over time, and the same configuration description (i.e. query) will not necessarily give the same result if it is re-evaluated later.

The configuration evaluation must be able to take into account the stability or validities. A configuration description can be evaluated into a valid or stable choice, or with no such constraint.

It should also be possible to evaluate a configuration description into a choice within the ambition for an edit task, and indeed to specify the ambition itself through the configuration description. One possibility would be to use the mandatory parts of the description to select the ambitions, and then the preferences are used to narrow the resulting ambition down to a single choice.
In the context of a software engineering database, we do not consider the last functionality quite as essential, though: It is reasonable to expect the persons in charge of making the changes (i.e. analysts and programmers) to have skills making them capable of handling the lower levels of options, validities and ambitions. Also, it can be argued that they need to have knowledge of this structure in order to efficiently integrate their work under different options. However, other persons who are producing configurations of the system for other purposes will not have the same detailed knowledge and skills as the programmer, but perhaps more domain knowledge instead. This could be personnel running tests, sales people configuring a customized system version for a customer—or even the customers themselves! These potential uses should be considered when the language for configuration descriptions are designed.

Since the result of the evaluation of a configuration description is a list of option settings, the evaluation process itself can be abstractly represented by a (pruned) decision tree, where each internal node represents the decision to set a specific value for one option. This may be used as a basis for efficient implementation of notifiers, see section 5.3.3.

Configuration descriptions: design sketch

Options are grouped into features, such that each option belongs to one or more features. Each feature has an enumerated set of possible values, representing mutually exclusive functionalities in the configured system, for instance $\text{OS} = \text{VMS}$ or $\text{OS} = \text{Unix}$.

Selecting a particular value for a feature implies that some of the component options are given definite values. Which options to turn on or off are determined by selection predicates associated with the features, in combination with constraints explicitly specified in the configuration description. In addition to the set of options, each feature has a default value to use if the feature is left unspecified.

For each value there is a selection predicate. In addition, the feature may be qualified in the configuration description, i.e. a qualifying predicate is attached to the feature value specification, e.g. $\text{OS=UNIX(date<010190)}$. When a configuration description specifies that a feature is to be given a specific value, the conjunction of the selection predicate and the qualifying predicate is used to select values for the options. In both cases, all value bindings for component options which can be inferred from the combined predicate are added to the choice.

The same process is repeated for each specified feature. If there is a conflict between two features, i.e. that they try to bind different values to the same option, an error is signalled. Optionally, the choice is checked for conformance with the validity, or the validity can be used to complete the choice.
5.3 Synchronization and transactions

As we briefly noted in section 5.2.3, the units of work in the change-oriented model, the so-called edit tasks, are essentially long transactions. For generality, we will actually distinguish between transactions and edit tasks. Just as an edit task, a transaction has an ambition defining which versions it may affect, but the transaction does not have a an associated choice. We allow each transaction to start one or more edit tasks: To start an edit task, a choice within the ambition is defined. The ambition for the edit task can also be a subset of the transaction’s ambition.

Nested long transactions come naturally in the change-oriented model: Subtransactions have ambitions contained within the parent’s ambitions. It is convenient to have a strict hierarchy, such that each subtransaction has only one parent. In this model, the edit tasks are the “leaf transactions” where the actual modifications of the database are performed.

We will now discuss more specifically how the change-oriented model works in cooperation with concurrent, multi-user access to the database.

5.3.1 Synchronization in the choice space

In a traditional multi-user database management system with locks and concurrency control, each process works within the context of a single transaction. For our purposes, the significant attribute of a transaction is the list of locks the transaction possesses, in particular the list of write (or exclusive) locks. We can imagine an object space—the space of all objects in the database. The purpose of the lock manager of the DBMS is essentially to maintain a partitioning of the object space into non-overlapping regions, one for each transaction, plus one for the read-locked objects and one for objects with no locks set. Each transaction is only allowed to modify objects in its own partition of the object space, and hence transactions cannot interfere with each other.

Change-oriented versioning, with its global options, introduce a separate choice space, which consists of all complete choices. The object and choice spaces are orthogonal. The ambitions and validities are just regions in the choice space. To fully determine the working context of a transaction, we now have to specify a region in the “cartesian product” of the object and choice spaces. Transactions can work on the same object, provided they work on different versions of it, i.e. in different regions of the choice space, see figure 5.10.

By requiring ambitions for different edit tasks to be disjoint, the ambitions (see section 5.2.3) can serve directly as such a partitioning of the choice space. Recall that the updates performed by an edit task are never visible outside its ambition. We can perform all synchronization in the choice space in this way, or we can also further delimit the accessible objects in the object space.

In that case, different transactions can work on the same version, but in a different subsystem. The synchronization problem is somewhat more difficult, because relationships may span arbitrary distances in the object space, and hence it is more
difficult to cleanly partition it into non-overlapping regions.

5.3.2 Overlapping ambitions

Non-overlapping ambitions correspond directly to traditional preemptive concurrency control (i.e. with exclusive locks) in database management systems. It is generally agreed that this is a too strict concurrency model for engineering databases with long transactions. We will instead use an optimistic concurrency control scheme where transactions may overlap both in the object space and the choice space (i.e. overlapping ambitions).

Changes made in overlapping transactions are visible in a common region of the choice space (shaded in figure 5.11) and must be coordinated. However, each transaction has an associated logical subdatabase containing the modified objects (versions), and the changes are local to the transaction until commit.

When a transaction is started up with a given ambition, its ambition is compared with the ambitions of the other active transactions. When there is an overlap, the transactions involved are notified, and several things may happen:

- The new transaction may wait for the older one to commit, effectively giving non-overlapping transactions.
The transactions may negotiate which and how much of the ambitions should be retracted.

- The transactions may continue unchanged, but are now aware that there is an overlap.

Which action to take depends on project-specific policies.

When a transaction with an overlapping ambition commits, all transactions it overlaps are again notified. The transaction receiving such a notification is expected to merge the updates the first transaction committed into itself, and will not itself be able to commit before it has done so.

5.3.3 The notification service

The notification service is a special purpose trigger facility. Conventional database triggers have a fairly general enabling predicate and a subset of the data manipulation language can be used to express the trigger action. Notifiers are triggered by one of a few special events in the database, and the only action directly performed is to send a message to the process requesting notification.

The first class of notifiers were discussed in the previous section: When two or more
transactions are running with overlapping ambitions, they are notified about the
progress of each other. The second class informs a transaction about changes to the
option structure which may be of interest to this transaction:

We assume that each transaction used a configuration description to select its am-
bition and choice, and that the transaction manager has recorded and associated
this configuration description with each transaction. When a transaction commits,
this results in some change to the options, their attributes and the validities. This
may affect the configuration another transaction has indicated it is interested in,
in that the associated configuration description would give a new choice if it now
was reevaluated. For instance, the configuration description might state that the
transaction is interested in new revisions as soon as they reach a given validity level.
The database system will notify the transaction when this happens. The transaction
can respond with two different actions:

- Mutate into a new transaction, with the new choice and ambition. Logically,
  this is a commit of the old transaction, (atomically) followed by the start of
  a new transaction, such that the new transaction keeps all locks and other
  resources the old one held.

- Refine the configuration description, such that the new configuration descrip-
tion will evaluate to the old choice and continue the same edit task.

As we pointed out, the configuration description also defines a subset of the objects
in the database, which the transaction is working on. This allows some filtering of
the raw notification events. At the bottom level, the option manager component
of the database management system will detect that the choice resulting from a
configuration description will change, i.e. some options change state between True,
False or unknown. The next higher level will check whether any object in the
configuration is actually affected, i.e. whether any visibility associated with any of
those objects changed from True to False or vice versa.

The final notification event as received by the activity will contain both the list of
options that changed and which objects these changes do affect.

5.3.4 Checkout/checkin

In order to isolate transactions with overlapping ambitions from each other, they
have to operate in different workspaces. One way of creating workspaces is by using
physically distinct areas (subdatabases) in the database. Objects or versions of
them are transferred between subdatabases by checkout and checkin operations (see
section 2.2.2). Since transactions are organized in a strict hierarchy, both the source
and the destination of checkout and checkin operations are unique subdatabases.
This in contrast to other long transaction models where objects can be checked out
to the same subdatabase from arbitrary other subdatabases.

In this context, there are two possible designs for transactions:
• "Full copy" model: Subdatabases contain copies of the entire region of interest from the parent database. Checkout is a physical copying of data from the parent. Only fragments with visibility intersecting the ambition are copied. On checkin, the contents of the subdatabase is compared to the parent's contents and the differences are incorporated in the parent, with appropriate visibilities.

• "Delta" model: Subdatabases contain only the differences from the parent database. Typically, some "copy-on-write" scheme will be used: If a fragment is modified, the new version is written into the local subdatabase and masks the old version in the parent.

In both cases, checkin (commit) means that the changes done in the child transaction are "pushed" upward to the parent and made visible to the parent and the siblings. For the first "full copy" model, the changes will not be visible for concurrent, overlapping sibling transactions, only to new sibling transactions started after the checkin. For the second "delta" model, changes will be immediately visible for all fragments which are not masked by local copies. Thus, if a transaction is dependent on some information being stable, it may copy those fragments to its local table prior to reading.

For additional concurrency control, we rely on timestamp-based techniques. With the "full copy" model, we can assume that both checkout and checkin can be done atomically (i.e. in a short transaction with strict concurrency control). We therefore have no problems with concurrent transactions committing their changes, with a checkout getting inconsistent versions of modified objects. Either all changes from an earlier transaction are present, or none of them. Only write timestamps are required. A potential update conflict is flagged if any objects/fragments in the parent database (the part which was checked out) have timestamps later than the starting time of the committing transaction.

With the "delta" model, objects/fragments are copied into the local database as they are read. Since objects are read at different times, there is the possibility that an intervening, committing transaction may cause inconsistent object versions to be read. For strict concurrency control, both read and write timestamps must be recorded, and the standard timestamp-based protocol is used [Ull88, p. 531].

If update conflicts are detected on commit, reconciliation (merging) will require applications operating in two transactions in order to examine the conflicting versions in the parent and the committing child. Change-oriented versioning does not permit those two versions to be somehow present in the same transaction, because there is no option by which to distinguish between them. (This is precisely why there is an overlap in the first place.) Keeping in mind that these are long transactions which application programs are able to explicitly connect to and disconnect from, this is permissible.

Strictly speaking, there is no need to go to the extra machinery of creating distinct subdatabases unless overlap does actually occur. However, if an overlapping transaction ever is started, and the old transaction had no local subdatabase, there will be no way to distinguish the changes local to the old transaction which logically are not committed yet. For the "delta" model, the fragments read will not be recorded
either, even though the transaction depends on them staying stable and would have copied to the local database otherwise. If this is of no concern, the creation of a subdatabase and physical checkout can be deferred until the overlapping transaction actually is about to be created. Otherwise, creation of new overlapping transactions must be restricted to those transactions which had created distinct subdatabases when they started. This leads to transaction modes vaguely resembling lock modes:

- (U) Unconditionally creates subdatabase.
- (C) Conditionally creates subdatabase (if overlap when started).
- (L) Lazily creates subdatabase when needed.
- (N) No subdatabase, no overlap permitted.

The compatibility matrix is given in figure 5.12.

---

![Compatibility matrix for transaction modes](image.png)

**Figure 5.12: Compatibility matrix for transaction modes**

The details of how differencing on checkin is done, respectively the delta representation, depend somewhat on the data model. Section 7.3.1 presents a concrete design for versioning of relational tables. The basic idea is that corresponding fragments in parent and child database are matched and their visibilities compared. "Corresponding fragments" are for instance relational tuples with the same keys. If the values and visibilities do not match (i.e. updates were made in the child), the update is done in a way that gives it exactly the same effect as if all the matching fragments in the database were deleted for the extent of the child’s ambition and the child’s fragments were inserted instead. If timestamps are used, the handling of timestamps has to be integrated with this differencing.

To simplify boolean expressions, it is possible to let visibilities and other expressions occurring in subdatabases be relative to the ambition of the corresponding transaction. As discussed below, we will restrict ambitions to be conjunctions of negated and non-negated options, i.e. lists of option/value bindings. On checkout, these option values can be directly entered in the expressions and the resulting expressions simplified.
5.3.5 Option-based synchronization

The mechanism for subdatabases and checkout/checkin presented in the last section is essentially independent of how versioning is done. An alternative mechanism exploits change-oriented versioning better. In this model, overlapping ambitions are internally disambiguated by options.

Each transaction $T$ is associated with a local option $O_T$, which is used to control the visibility of its own changes. The local option is introduced with the transaction, but as long as $T$ has not committed, the default value of $O_T$ is True only in $T$ itself. For all other transactions, unless $O_T$ is explicitly specified, it remains False and the modifications done in $T$ remains invisible.

After $T$ commits, the default for $O_T$ changes to True for all new children of $T$’s parent $P(T)$. (More precisely, in all transactions where the parent’s own local option is selected, i.e. we specify that the parent’s local option implies the local options of all its committed children.) Note that it is the default which does change: In other transactions which did not select $O_T$, $O_T$ remains False, and these transactions will not see the changes committed by $T$ before they elect to do so by setting $O_T = True$.

The remaining transactions with $O_T = False$ have done changes which would be discarded if they just committed, since the choices with $O_T = False$ will be removed. Thus, it makes sense to require them to merge their changes with the new changes under $O_T = True$ before committing. This could be done by disallowing commit for transactions with $O_T = False$ and providing a mechanism for removing $O_T$ from the ambition. A transaction which finds it cannot commit, because it has an outdated value (False) for some option $O_T$, removes $O_T$ from its ambition and thus “imports” the changes described by $O_T = True$ into its own environment. After having acknowledged those changes in this fashion, it can merge its own changes and finally commit.

Eventually, all sibling transactions which were running concurrently with $T$ or otherwise declined to see $T$’s changes will terminate. At that point, no more transactions in the subtree starting with $P(T)$ will have $O_T = False$. We can then eliminate $O_T$ by incorporating it in $O_{P(T)}$. We replace all occurrences of $O_T$ with $O_{P(T)}$ and thus eliminate $O_T$ from all expressions.

The problem of the granularity of the inter-transaction interactions remains. Some of the virtue of a long transaction mechanism with checkout and checkin is that it is natural in a distributed environment. But in a distributed environment, it is not possible transparently to propagate changes between transactions at an arbitrary granularity. As a trivial example: If two transactions are editing the same file concurrently, it is not feasible to echo the effect of each keystroke in one transaction to the other transaction immediately. We are not making the point that this is generally useful, merely that there is no transparent way to hide the fact that synchronization between transactions by necessity happens at a certain granularity.

In the previous “copy-oriented” model changes are propagated between transactions by explicit copy operations (some of them as part of commit). In essence, we do not hide the explicit synchronization at all, and instead elevates it to a necessary
and meaningful element in the model. But with this “option-oriented” model, any
transaction could asynchronously select the local option for another transaction and
from then on expect to see the changes done in the other transaction. It is necessary
to supply an explicit synchronize operation—which is an alien element in this
model, and in a sense brings us back to where we were, with the problem of conflicting
updates etc.

5.3.6 Primary ambition

The synchronization model can be refined by introducing primary ambitions, see
figure 5.13. The primary ambition is a subset of the full ambition. The full ambition
is still used to compute visibilities.

The full ambitions of two transactions are permitted to overlap each other, but
overlap is not permitted for the primary ambitions. Thus, a transaction may reserve
the right to operate on a specific area of the choice space for itself, but still elect
to make its changes more widely visible. The choice of a transaction is constrained
to be within its primary ambition, and for subtransactions, their ambitions must
be within the parent’s ambition and their primary ambitions within the parent’s
primary ambition.

In this case, it probably also makes sense to require the task stability for the trans-
action to be a subset of the primary ambition.

![Diagram of Ambitions and primary ambitions](image)

Figure 5.13: Ambitions and primary ambitions

5.4 Complexity of Change-Oriented Versioning

There are a number of complexity issues to consider with respect to change-oriented
versioning. Most of them originate from our heavy use of symbolic boolean algebra
above, and the expressions must be simplified in order to keep the complexity man-
ageable. Frequently, we are also interested in whether expressions are satisfiable.
Optimal simplification and satisfiability of boolean expressions are NP-complete problems, but in none of the cases above do we depend on optimal or absolutely precise solutions. Our general approach will be to use heuristic algorithms, and then make conservative decisions when these algorithms fail to give precise information. However, we shall see that it is possible to make simplifying restrictions on the form of the expressions. This will allow us to determine satisfiability with a polynomial procedure in the cases where efficient evaluation and precise determination is crucial.

5.4.1 Visibilities

The main "administrative overhead" for a change-oriented versioning tool will be the storage of visibility expressions. Visibilities will be stored in a separate table or file, indexed with a small integer, and only these integers will actually occur in delta files or the database proper. However, there are some indications that the visibilities will be many and complex, and that the storage and manipulation of visibilities might be a major overhead.

There is no need for optimal simplification of visibilities in order to correctly reproduce versions which have earlier been stored in the archive. To do so, it suffices to set values for the options and evaluate the visibilities directly, a process which is simple and linear in the accumulated size of the visibilities.

The sources of growth in the number of visibilities are:

- For each edit task which inserts new fragments, a visibility equal to the task's ambition is created.
- Within a task, each time a fragment with a distinct visibility is modified, one or two new visibilities are created.

We can make a simple argument to the effect that the number of visibilities at least asymptotically will grow linearly in the number of edit tasks. We have to make some assumptions about the distribution of visibilities and changes over the fragments. In the first approximation, we believe that independent changes are spread evenly through the file, and will not have a tendency to cluster on the same lines.

A more operational formulation of this assumption is that the number of visibilities affected (split) is proportional to the number of lines changed and the "density" of visibilities on those lines (i.e. the total number of visibilities divided by the total number of lines in the archive). The other assumptions are:

- The number of lines in a version is constant, \( l \).
- There is a fixed probability \( p \) that a given line in a version is going to be changed by an edit task.
- For each changed line, \( k \) new visibilities are created. \( k \) is 1 or 2, depending the rule for updating visibilities.
Let $s_e$ be the total number of visibilities after $e$ edit tasks. Since each version is $l$ lines long, each task modifies $ql$ lines. Consequently the total number of lines in the archive after $e$ tasks is $l + ep = (1 + ep)l$.

Each task therefore checks out a fraction $1/(1 + ep)$ of the total number of lines in the archive, and by assumption is potentially affecting the same fraction of the total number of visibilities, $s_c$.

Of the lines checked out, and by assumption, the visibilities on these lines, a fraction of $p$ is modified. The fraction of the total number of visibilities which are attached to modified lines is $p/(1 + ep)$. For each of those, $k$ new visibilities are created.

Ignoring for the moment that the number of visibilities is integral, we thus get the number of new visibilities in edit task $e + 1$:

$$\Delta s = s_{e+1} - s_e = \frac{ksep}{1 + ep}$$

A little manipulation yields

$$r(e + 1) = s_{e+1}/s_e = \frac{1 + (k + e)p}{1 + ep}$$

Let $q(i) = 1 + (i - 1)p$, thus $r(i) = q(i + k)/q(i)$. To get a closed formula for $s_c$, we set $s_0 = 1$, which gives:

$$s_e = \prod_{i=1}^{e} r(i) = \frac{\prod_{i=1}^{e} q(i + k)}{\prod_{i=1}^{e} q(i)}$$

By cancelling terms and adjusting the indices

$$s_e = \prod_{i=1}^{k} \frac{q(i + e)}{q(i)} = \prod_{i=1}^{k} \frac{1 + (i + e - 1)p}{1 + (i - 1)p}$$

$k = 1$ gives $s_e = 1 + ep$, i.e. linear growth. Intuitively, under these assumptions, the average number of visibilities to add in each round is constant, because the number of the lines to spread the visibilities over grows as fast as the number of visibilities themselves.

$k = 2$ gives $s_e = \frac{(1+p)+(2+p)e+p^2e^2}{1+p}$. I.e., the number of visibilities rises quadratically, although the coefficient of the quadratic factor is small ($p^2$).

Since we have ignored that the number of visibilities is a discrete quantity, these estimates are only valid when $\Delta s$ is large enough that the rounding error is small in comparison. For the $k = 1$ case, $\Delta s$ is constant. In fact, at least 1 (or 2) visibilities are added for each edit task, so the proportionality factor is at least as large as that, not $p \ll 1$. For the cases $k > 1$, the estimates are at least asymptotically valid, since $\Delta s$ is increasing. We have also ignored the effect of straight insertions. Any edit task that inserts new lines (as opposed to modifying old ones) will create one new visibility equal to the ambition.
We should also point out that the length of the new visibilities will also grow. If the length of the ambitions on the average stay the same, the length of the longest visibility will grow linearly. However, it is likely that ambitions will grow somewhat more complex as the number of options grow.

For a worst case estimate, it is reasonable to assume that there is a bounded number $F_{\text{max}}$ of fragments in any version, and that this bound is constant over time. The upper bound on the number of new visibilities per edit task will then be $2F_{\text{max}}$. However, with fine-grained deltas $F_{\text{max}}$ is large: the number of characters in the largest file version, or something similar. Even though the worst case is linear, the proportionality factor is ridiculously large.

There is a possibility that new visibilities which are created during an edit task are equivalent to already existing visibilities, or are equivalent to False. In this case, there is of course no need to add to the tables, but these cases cannot be reliably detected, since it is not tractable to simplify the expressions to a canonical (optimal) form. It also remains to be seen if this happens frequently enough in practice that there is any use in trying to detect it.

As a concluding comment, let us note that many and complex visibilities also indicate that the versioning structure one is attempting to represent is complex—perhaps too complex for the developer to comprehend anyway! Consider as a boundary case a hypothetical ideally modularized program system, where each optional feature is strictly localized to one or a few components. The set of options is partitioned into groups of a small, bounded options used to select one out of a handful of alternatives for each of the features. In this case, the upper bound for the number of visibilities grows approximately linearly with the number of orthogonal features!

The essence of this is that there is a tendency for distinct options to affect different parts of the source text (or whatever else is versioned). Taking this into consideration, our estimate above, which assumed that the location of changes were independent of each other, and which lead to quadratic behavior, might actually turn out to be too pessimistic. Any more precise estimation needs data on update frequencies and clustering of updates on fragments—depending on application, programming practices, skill etc., and we fear that would be so speculative at this point that the estimates remain useless.

### 5.4.2 Checkout

Given a choice which determines all relevant options (those occurring in visibilities in the region of interest in the database), checkout is simple: Visibilities are evaluated, using the values for the options determined by the choice.

However, a choice could be complete, even if it does not determine all options. To determine whether this is the case is NP-complete, since it involves determining satisfiability for each visibility: The choice is complete iff for all visibilities, either the visibility itself or its negation is non-satisfiable.

The consequences of not being precise here is small, since the user could easily add
options to the choice until the system is able to determine that the choice is complete. Alternatively, we could require choices which determine all relevant options in those cases where complete choices are required.

For cases when a multi-version database corresponding to an ambition is to be checked out, all fragments with a visibility intersecting the ambition have to be copied. For practical purposes, it is not critical to compute the intersection precisely. If it is not possible to determine that the intersection is empty, the fragment has to be checked out, and the only loss is the increased size of the subdatabase.

### 5.4.3 Ambitions

In the discussion above, we have largely assumed that ambitions are arbitrary boolean expressions. We believe that this is excessive generality with marginal benefit for the user. Instead, we will restrict ambitions to be a conjunction of negated and non-negated options, i.e. an under-specified choice.

This will make testing for transactions with overlapping ambitions simple and fast. Some simple bitmask operations suffice: Each ambition is represented by two bitmasks, $\text{Def}$ with a 1-bit for each option defined by the ambition, and $\text{State}$ with a 1-bit if the corresponding option is negated. To test whether ambitions $A$ and $B$ overlap, compute $\text{Dis}_{AB}$ according to figure 5.14. The 1-bits in $\text{Dis}_{AB}$ indicate which options have different status in the two ambitions, and thus keep the ambitions separate. In other words, if the result is non-zero, the ambitions are disjoint. The empty set (the boolean expression $\text{False}$) cannot be represented in this scheme, and therefore, intersection is undefined if the sets are disjoint. The set of representable sets is not closed under union. Thus, no union operation is defined.

\[
\text{Dis}_{AB} = \text{Def}_A \land \text{Def}_B \land (\text{State}_A \oplus \text{State}_B) \tag{5.8}
\]

Intersection $I$ (only if non-disjoint):

\[
\text{State}_I = \text{Def}_A \land \text{State}_A \lor \text{Def}_B \land \text{State}_B \tag{5.9}
\]

\[
\text{Def}_I = \text{Def}_A \lor \text{Def}_B \tag{5.10}
\]

---

**Figure 5.14:** Computing set operations on ambitions

Having choices and ambitions on the same form also seems to make it simpler and more natural to determine both through the same high-level mechanism (configuration descriptions).

We assume that the ambitions normally will leave relatively few options unspecified. The reason is that the number of different versions the user has to juggle in an
edit task grows exponentially with the number of unspecified (free) options. This will simplify some of our problems somewhat. In several cases, we need to test whether an ambition intersects another set. The case that this set is an ambition was discussed above. If the set is represented by a characteristic predicate which is an arbitrary boolean expression, the intersection with the ambition is computed by inserting each option determined by the ambition. Thus, the upper bound on the number of variables in the resulting expression is the number of free options in the ambition. If that number is small enough, it is still within our reach to determine whether the expression is satisfiable.

5.4.4 Stability

To implement stability, ambitions have to be checked for overlap with the stability predicate. Such tests are done by checking if the conjunction of the predicates is satisfiable, i.e. a NP-complete problem in general. If that is the case, we would have to reject ambitions which the heuristic simplification algorithm cannot determine does not overlap with the stability, even though there is in fact no overlap. Depending on which form we manage to keep the stability predicate on, we should be able to do better than that:

The stability will be a disjunction of stability contributions from distinct tasks (task stabilities):

\[ V_0 = \bigvee_i V_{0i}^T \]

Thus, it is sufficient to test for overlap with the individual task stabilities (\( V_{0i}^T \)). If we restrict the task stabilities to the same form as the ambitions, this is simple. (Recall that ambitions have a simple form which permits fast disjointness tests.)

Even if the task stabilities have a more general form, we know that the task stability is contained in the task ambition. If the ambition is saved along with the task stability, we can rapidly screen out those task stabilities where we cannot possibly have any overlap. For the same reason, we know that the options occurring in the task stability are restricted to those left unspecified in the ambition. When testing, we are intersecting with another ambition, which might determine even more of the options. As mentioned above, we assume that the ambitions leave relatively few options unspecified. Hence, even if the task stabilities have a general form, the expressions will involve few options, and a heuristic simplification algorithm might determine satisfiability with enough precision.

Regardless of what form the task stabilities take, further speedup can be gained by noting the history of introduction of options. For each task \( i \) before the option \( O_j \) was introduced, the term \( \neg O_j \) is multiplied into the task validity. In other words, if the ambition contains \( O_j \) non-negated, there cannot be any overlap with task stabilities for task performed before \( O_j \) was introduced. Hence, it suffices to test the task stabilities for tasks younger than the youngest non-negated option in the ambition.

It is also worth noting that the task stabilities will be mutually disjoint, because each task stability is contained within the ambition of the task which produced it,
and the ambitions never overlap previous stabilities.

5.4.5 Intersecting visibilities

In some cases, it is interesting to determine whether two visibilities intersect, mainly to verify that two fragments tagged with these visibilities will never turn up in the same version. One case is for instance insertion of a new tuple in a relational table. If key uniqueness is to be maintained, it is necessary to verify that no existing tuple with the same key has an intersecting visibility\(^2\). Another case where this may be needed is when checking in a multi-version database when a nested transaction is committed. For some transaction implementations, it is necessary to compute a multi-version difference between the database states in the subtransaction and its parent.

Directly testing for disjointness is intractable in general, because visibilities do not have any special form, such as ambitions and stabilities have. Fortunately, in all the important cases we have in mind, one of the visibilities is for a new fragment, and thus is equal to the ambition of the task inserting it. As we discussed above, testing for intersection with an ambition is tractable, provided the ambition has few enough undetermined options.

Even if that number is too large to permit application of an exact algorithm, we have a workaround for those cases where a heuristic algorithm fails to give precise results (i.e. does not manage to simplify the intersection to \textbf{False}): If there is a potential overlap between the ambition \(A\) and an old visibility \(S\), this visibility can be modified to \(S \land \neg A\) to ensure disjointness. The cost of this is that we get spurious new visibilities which may in fact be equivalent to old ones, and the increased number of visibilities could in some cases be significant. If all insertions need such checks, if the number of conflicting fragments increases with the number of edit tasks performed, and if splitting has to be done in most of these cases, we might in the worst case end up with an exponential growth.

The case of multi-version differences alluded to above will also require determining intersections between visibilities. It is difficult to discuss this concretely without taking the data model into account.

In some cases, in particular when tracking derivation dependencies, it is useful to compute the intersection of a number of visibilities. For instance, the set of choices for which a version of an object is the same is given by the intersection of all the visibilities that evaluate to \textbf{True} under the current choice. The set of choices for which a derived object version stays the same is the intersection of the visibilities of the visibilities of all objects examined in order to derive it. Thus, this expression could be explicitly set as the visibility when storing the derived object version. Such visibilities might become very long and complex, but they can safely be simplified in a way that removes choices:

\[\bullet\] If the expression is in the form of a disjunction of terms, top-level disjuncts

\(^2\)Maintenance of key uniqueness is discussed more generally in section 5.6.3.
may be removed.

- Options can be set to specific values.

To what extent this is done is a tradeoff between increased work in handling the expressions vs. increased work in (strictly unnecessary) rederivations.

Computing some relational algebra operations (e.g. joins) on tables with visibilities on the tuples will also involve computing new visibilities which are the intersection of visibilities in the input tables. Some of these intersections may be empty, and to keep the sizes of the output tables as small as possible, it is important (but not critical) to be able to detect that the expressions simplify False. More on this in section 5.6.3.

5.4.6 Validities

We have not made many assumptions on the form of the validities. They might even be represented in a form different from boolean expressions, for instance first-order logic. In this section, we are still considering only validities which are boolean expressions.

Validities are mainly used for those two purposes:

- Testing specific versions when they are checked out of the archive.

- Completing non-unique choices to get a complete choice which is valid according to a specific validity.

The first use is non-problematic, assuming that the choice determines all options. The validity is simply evaluated with the values for the options given by the choice. If the choice is complete without determining all options, and the validity cannot be simplified to True or False, the only safe conclusion is that the choice is invalid. It might be that the validity really is equivalent to True or False, but the heuristic algorithm is unable to determine that. However, it is also possible that the validity is undertermined. If the choice does actually cover several points in the choice space, those may have different validity. In this case, it will be a matter of definition whether the choice is valid.

With validities there are some other complexity-related issues. Most important is the need to express mutually exclusive options (for instance representing an enumerated value). If this constraint is to be represented as a boolean expression, the length of the expression grows as the square of the number of mutually exclusive options. This means that there is a severe limitation on how many options such a set may contain.

There may be other constraints which could more easily be expressed in a more expressive language, for instance first order logic. Using a general language like that is still not likely to reduce the complexity of testing the constraints. A more specialized language (for instance with MutuallyExclusive(O_1, O_2, ...)) as a primitive) might be
better, but no design has yet been made. For the time being, a representation as
boolean expressions is still best suited, in spite of the complexity issue. The rea-
son that keeping all predicates on the same form and using the same simplification
procedures permits us to use validities to simplify other predicates. This could be
extremely difficult to do if different representations were used.

5.4.7 Cutting down the complexity

Deleting versions

It is possible to reduce the complexity of the expressions by throwing away old
versions.\textsuperscript{3} Some options may be outdated or experimental, and the versions selected
by one of the values of the option are no longer needed. Such options may be set to
constant values and the expressions simplified accordingly. It may not be possible to
reduce the number of validities very much in this way, because it is impossible in a
general way to determine for each validity after simplification whether it happens to
be equivalent to some other validity. However, some of the validities may simplify
to \textit{False}, and the corresponding fragments may be removed from the database, if
the time needed for the compaction is warranted.

Limiting the scope of options

Another possibility for keeping the complexity down is somehow to reduce the scope
of the options, namely limit the area of the database where each option is permitted
to influence versions, i.e. occur in expressions. The complexity of expressions do
not depend on the total number of options, but only the total number of options
permitted to occur in the same expression. If for instance the database is partitioned
into distinct subdatabases, with disjoint sets of options controlling the variation in
each of them, complexity will be significantly reduced.

The global consistency constraints are still possible to express through "interface
validities" between pairs of partitions. A possible design would be to organize the
subdatabases in a tree (a spanning tree over the composition structure of the product,
for instance), and to set up interface validities for each parent/child pair. Each
interface validity expresses which combinations of choices in the parent and the child
are permissible. The interface validities are the only expressions permitted to refer
to options in more than one partition. Each partition has its own set of validities,
validities, stability etc. Each transaction has a separate choice and ambition for each
partition it is actually accessing. With this setup, the complexity of the expressions
is limited by the number of options in each partition. If the tree structure reflects
the composition structure of a software system, note how this compares to a well-
modularized system in the traditional sense.

\textsuperscript{3}Before throwing away old versions, it is of course possible to take a snapshot of the database
to tape or similar.
5.5 Comparison with Traditional Systems

5.5.1 Emulating a version-oriented model

The change-oriented model is capable of representing the version structures in any of the common traditional (i.e. "version-oriented") models. In these, individual versions within a version group can be thought of as vertices in a so-called version (or revision) graph. The main differences are in what structures are permitted for this graph:

- A flat space of versions without any mutual relationships. This corresponds to a graph without any edges.
- The most common case (SCCS, RCS etc.) is that the version graph is restricted to a tree.
- In the most general case, the version graph could be an arbitrary directed acyclic graph. (I.e. version merging is incorporated in the model.)

Any version graph can be represented within the change-oriented model:

- If the graph is disjoint, add a node (representing an empty initial version), and arcs from this node to the roots of all the disjoint portions of the graph.
- Introduce an option corresponding to each node in the graph.
- The edges in the version graph are expressed through the validity:
  
  - For each edge from a node with option $O_1$ to a node with option $O_2$, add the term $O_2 \Rightarrow O_1$.
  
  - For each pair of nodes with options $(O_1, O_2)$ immediately succeeding a fork in the graph: Let $\{J_1, \ldots, J_k\}$ be the set of options for all nodes where distinct paths originating from $O_1$ and $O_2$ are joined (i.e. version merges). If this set is nonempty, add the term

    $O_1 \land O_2 \Rightarrow \bigvee_{i=1}^{k} J_i$

    If no such nodes exist ($k = 0$), add the term $\neg(O_1 \land O_2)$, expressing mutual exclusion between nodes on divergent revision branches.

For the version graph in figure 5.15, the corresponding validity is determined as follows: To version $Vi$ in the version graph, there is a corresponding option $O_i$. For the arcs, we get

\[(O_6 \Rightarrow O_4) \land (O_6 \Rightarrow O_5) \land (O_5 \Rightarrow O_3) \land (O_4 \Rightarrow O_3) \land (O_3 \Rightarrow O_1) \land (O_2 \Rightarrow O_1)\]

Strictly speaking, this is enough information to be able to do option selection. Instead of selecting a version in the version graph, the user will select the corresponding
option. The implications in the validity will then determine which other options to select (namely those on the path backward to the initial version).

In order to explicitly disallow mutually exclusive options (like $O_2$ and $O_6$ for instance), we have to take into account branches:

$$\neg (O_2 \land O_3) \land ((O_4 \land O_5) \Rightarrow O_6)$$

The final validity is the conjunction of this and the previous expression.

In the change-oriented model, options and validities are global for the entire system, while most version-oriented models have a version graph for each component in the system. If this is the case, we will have to repeat the steps in the last paragraph for each component, and we will get $O(\text{versions} \cdot \text{components})$ options. This is a worst case estimate, and in practice it is not possible to orthogonally choose component versions. For instance, versions of one component are paired up with corresponding versions of another. That is, some versions of different components can be described by the same option. (Recall that a main rationale for the change-oriented model is just that corresponding changes in several components can be described by a single option, since options are global.)

### 5.5.2 Change-oriented vs. version-oriented versioning

In the introduction to this chapter we compared change-oriented versioning with more traditional, version-oriented versioning models. In section 3.1.1, we defined version-oriented versioning models as those where object versions themselves are explicit objects, and thus can be separately named, independently from the other objects. In contrast, change-oriented models are characterized by having version selection orthogonal to object selection, and versions selected as the composition of optional sets of changes across all objects.
In change-oriented versioning, selection of mutually compatible versions of the objects in a configuration is inherent in the model. This does of course not happen magically, but only when developers consciously make it happen. For version-oriented versioning, configuration management mechanisms have to be built on top of the versioning mechanism to record compatibility constraints between the object versions. Bound configurations, like DSEE bound configuration threads or Cedar system models (see chapter 4), explicitly record object versions making up a configuration which was at some time produced. This is safe, but very inflexible.

To be able to produce consistent configurations by combining object versions in new ways, more subtle compatibility constraints must be recorded. To this end, Adele and other systems attach attributes to object versions and express compatibility constraints by predicates over attributes. The developers have a way to express their knowledge of which object versions are intended to or proven to be compatible, which versions are definitely incompatible and so on, without explicitly enumerating all those combinations. The attributes play a similar role to options in the change-oriented model. Both characterize functional properties of the objects across all objects, and thus serve to express inter-object compatibility constraints. The difference is that COV has independent mechanisms to produce the configurations (setting a choice) and to characterize the consistency of them (validities).

The deepest difference between the change-oriented and the version-oriented approaches is in their view of the development process. With COV, there is a strong connection to the development process. The purpose of an edit task is considered to be to implement a part of the combined functionality of the product, expressed through the associated ambition. Ideally, the developer's intent about how the work in progress is to be combined with other work is captured from the outset.

In VOV, the focus is more strongly on the history of development of individual objects. The main version structure is the version graph, and the purpose of an edit task is to derive a new version from one or more old ones. But maintaining consistency across a number of objects basically means that several of these development steps really are part of the same task. Moreover, when objects have orthogonal functional properties, multiple parallel development steps for the same object may actually belong to the same task. These are the kinds of relationships that VOV have difficulty in expressing.

5.6 Application to Common Data Models

The previous sections presented and discussed the basic concepts of change-oriented versioning. This model can be applied to any of the data models in common usage for database management systems. We will describe how this can be done for a few particularly interesting cases.

The main interface of change-oriented versioning to the DBMS is through the visibilities. The visibilities define which parts of the database which should be presented to the client program, and which should be hidden from it; i.e. what version the client program sees. The visibilities themselves can, in principle, be determined
independently from the database.

The approach taken in the following discussion is to integrate the versioning with the concepts in the data model itself. The alternative approach, which we shall not discuss further here, is to apply versioning to the file system or record oriented layer that forms the foundation of most database management systems. In that case, the basic storage is versioned, but these services presents the same interface to the upper layers of the DBMS, except that each process accessing the database must specify which version to access (i.e. a choice). While conceptually simpler, the main drawback of this approach is that it hides the versioning too much from the application layer: It is not possible to relate versioning to the application level objects in any useful way, and in particular it does not appear to be possible to present a practical interface which allows an application to work on several versions in parallel.

5.6.1 Applicability of COV

In section 3.1.3 we discussed the applicability of version-oriented versioning. We noted that it is crucial to be able to form version groups, and that this is not possible for atomic objects and objects without identity. In particular, this creates severe problems for versioning of relationships or links in graph-oriented data models.

Change-oriented versioning does not have any of those problems. On the conceptual side, this is because the entire database is versioned uniformly, so it is the entire database—a complex object with identity—we are forming version groups of. On the technical side, it is possible to attach a visibility to almost any piece of information in the database, including links and relational tuples. For this reason, change-oriented versioning can be incorporated much more transparently in a database, essentially without affecting the data model.

5.6.2 Text files

Even though it is trivial as data model considered, to use text files is still the most common way to archive data (at least in software engineering environments, where our emphasis is). Holager[Ha88] does only consider change-oriented versioning of text files, and treats that in some detail. In particular, he proposes the “neighbor criterion” to determine the validity of an edit task.

Options, visibilities and validities are global across a collection of text files. In the simplest way, a file is versioned by prefixing each line in the file with a visibility. In practice, having text lines as fragments is too coarse-grained: Most changes are only affecting a few of the characters on the line. It is important to minimize the size of differences (i.e. alternative fragments), since this reduces the likelihood of merge conflicts, where changes done under different options affect the same portion of the text in conflicting ways. Therefore, we will relax this to allow arbitrary text blocks as fragments, from a few characters, up to the full size of the file.
This is not significantly more difficult to implement that uniformly line-sized fragments. It will be fairly easy to minimize the line-by-line deltas output from standard difference generators like the Unix tool `diff`: A postprocessor can compare changed lines and exclude the parts of them that are equal from the delta. The result will not be provably the minimal character-by-character delta (if indeed the tool does produce the minimal line-by-line delta at all), but this does not really matter. Measured by the characters involved it is still no worse than the line-by-line delta, and probably significantly better.

**The neighbor criterion**

Holager's neighbor criterion is one of the possible heuristics to determine the validity of an edit task. To define this criterion, Holager introduces the validity of a single edit command. (An edit command is a single insertion or deletion, as in the delta between the new and the original version.) Intuitively speaking, the edit command is valid for all versions (i.e. choices) where the text fragments immediately before and after the affected text remains the same as in the version the user was actually editing and checked in. The task validity is the intersection (i.e. conjunction) of the validity of all the edit commands this task consists of. This criterion is of course approximate, but is still an aid to the user both in determining which option combinations are likely to be valid and in pointing out the potential trouble spots. It is a somewhat stronger supplement to the merge conflict criterion utilized by RCS and SCCS. The neighbor criterion is treated fully in [Ha88] (sections 2.3, 2.4, 4.2 and 4.3 especially), which in particular contains an exhaustive analysis on how to determine the validity of each editing command, in each possible situation and in face of insertions, deletions and also text moves.

**Text moves**

The simple picture of text blocks prefixed with visibilities is complicated somewhat by the introduction of text moves. This is important to handle in practice, but since it is not required for the understanding of the other parts of this chapter, and is treated fully in [Ha88, chapter 4], we will skip lightly over it here. Suffice to say that it is handled by inserting markers, prefixed with visibilities, into the text. The markers occur in triples, marking the beginning of one, the boundary between, and the end of the other text block for two text blocks. Depending on the visibility of the move markers, the blocks are presented in the same or the opposite order in the resulting text version. The overall effect is equivalent to the notation commonly used in proofreading to indicate that two words are to be exchanged, i.e. \(\text{this}\) \(\text{like}\).

**5.6.3 Relational model**

The simplicity and regularity of the standard relational model [Cod70] allows for an intuitively simple and probably fairly efficient application of the change-oriented model.
We introduce visibility as a new built-in domain (i.e. attribute type). A relational schema is extended to a versioned relational schema by extending each table\(^4\) with a visibility attribute. For each table, a new key is defined, consisting of the old key (the user key) as its most significant part and the visibility as the least significant part.

A single version is produced by performing a select and a project operation on each table: First the tuples with visibility evaluating to True are selected, and then the visibility attribute is projected away. In a practical system, these two operations could be buried away inside the database management system.

For database updates, we have to be somewhat more strict about the visibility update rules. The reason is that we want to be able to guarantee uniqueness of keys in all possible versions of the database, not only the valid ones. (It is discussed below why we do this.) To recapitulate the rules from section 5.2.3:

- **Deletion:** Previous visibility \( S \) becomes visibility \( S' = S \land \neg A \).
- **Insertion:** New visibility \( S'' = A \).
- **Update:** Split tuple in two tuples with visibilities \( S' = S \land \neg A \) and the old contents and \( S'' = S \land A \).

The important property of these rules, is that they imply \( \neg(S' \land S'') \) wherever a tuple (i.e. key) is split.

In the case where a batch checkout/checkin paradigm is used, the deltas are computed based on the keys of the tuples at checkin time.

**Uniqueness of keys**

Uniqueness of keys is the main consistency constraint in a relational database. Since different versions of the database may have tuples with the same user key, but different values otherwise, we have added the visibility to the key. This fixes the problem from a restricted technical point of view. However, to guarantee uniqueness of key for each possible version of the database, we have to verify for each user key that the visibilities this key is occurring with are mutually disjoint.

More formally, for each user key \( K \), let \( \{S_1, \ldots, S_k\} \) be the set of visibilities of the tuples with key \( K \). Then we must have

\[
\bigvee_{i,j \in \{1 \ldots k\}, i \neq j} S_i \land S_j \equiv \text{False} \quad (5.11)
\]

Clearly, it is not feasible to check this condition for each database operation. However, if we maintain the condition at all times, the only case when we actually have to check is when a tuple with a new key (for this version) is added:

\(^4\)We will use table instead of relation to avoid confusion with other uses of the word relation.
Obviously, deletions of keys in the new database version can never result in new overlaps between keys, since this is handled by further restricting the visibility on the corresponding tuples. Let us consider insertions and updates.

For each tuple in the new version we are about to check in, and its user key, there are three cases:

- This tuple was never touched. According to the hypothesis, there is no other tuple in the new version with the same key, and therefore no operations were performed affecting this key at all.

- This tuple was updated. One tuple with visibility \( S \) is replaced with two tuples with disjoint visibilities \( S' = S \land \neg A \) and \( S''' = S \land A \), respectively. Since both imply \( S \), and for all \( j \), \( S \land S_j \equiv \text{False} \) they cannot result in new overlaps.

- This tuple is new. There are two subcases:
  - In the trivial case the key did not previously occur in the database at all.
  - A tuple with the same key occurs in some other version of the main database. In this case we have to check whether the visibility for this tuple overlaps any of the visibilities for the other tuples with the same key. We compute the union of the other visibilities \( S_i \), and check whether the intersection with the visibility \( S \) for the new tuple is empty, i.e.

\[
(\bigvee_{i=1}^{k} S_i) \land S \equiv \text{False}
\]  

(5.12)

The last case is one of the cases which was discussed in section 5.4, where the general case is NP-complete, but the simplified form we are using for the ambitions allows the problem to be solved for the normal cases.

As a final observation, we can even avoid this problem by modifying the visibility update rules further: Replace the ambition \( A \) with the task stability \( V_0^T \), and note that the effect of this is that the non-stable versions of the database now will be empty. (No tuple will ever be given a visibility which is not entirely within the stability after the edit task is completed.) Consequently, for the last subcase above, the visibilities \( S_i \) for the old tuples with the same key must all be contained in the old stability \( V_0 \). But \( V_0 \) is by definition disjoint from the ambition \( A \), which contains the visibility \( S (\equiv V_0^T) \) of the new tuple. Hence equation 5.12 always holds.

The cost of using this approach is precisely that the non-stable versions of the database are empty. Any edit task must start from scratch! Arguably, a number of the advantages of change-oriented versioning are lost.

**Normalization**

Even though we start from normalized non-versioned tables, after extending the tables with the visibility attribute, there is usually room for further normalization. In the simplest case we have some attributes which are essentially non-versioned, i.e.
their values do not depend on the visibility attribute, only the non-visibility part of the key. The table should be normalized by separating the non-versioned attributes into a table of their own.

Having thus disposed of the non-versioned attributes, we turn to the versioned attributes. The visibility, from the point of view of the relational model, is really a composite key. We can, in principle, have dependencies on parts of the visibility. If that is the case, we can (conceptually at least) split the visibility attribute and proceed with normalization in the usual way.

Such dependencies may be difficult to find, even if they are present. As a limiting case, the tables could be decomposed into one relation per attribute. This would in most cases result in severe performance losses.

**Relational algebra on versioned tables**

It is possible to perform relational algebra operations directly on versioned tables. For some of the operations this is trivial: Selection, projection and union are all performed just as in the non-versioned case. For projection and union, duplicates must in addition be removed, which is slightly different for versioned tables:

There is a possibility that distinct tuples with identical values but different visibilities turn up. These should be detected, and replaced with one tuple with visibility equal to the union of the original visibilities. To do so may not be strictly necessary, if the duplicates are removed when a particular version is produced. Still, the procedure is not more time-consuming than non-versioned removal of duplicates, and the sizes of the tables are reduced.

The two remaining operations for a relationally complete language, difference and cartesian product, require set intersections to be computed. Some of these intersections may be empty, and the corresponding redundant tuples will remain in the tables unless the boolean simplification procedure is able to reduce the corresponding expressions to False. In general this is not the case (see section 5.4), but fortunately correctness does not depend on it. However, the tables could grow impossibly large, depending on how complex the version structure is.

To compute the cartesian product of two versioned tables, compute the cartesian product as if it was a non-versioned, i.e. including the visibilities as ordinary columns. Then, tuple for tuple, the two original visibilities are replaced by a single visibility which is intersection of the two. The new visibility is the set of choices for which both the two original tuples were present in their respective tables. Some of these intersections are empty, meaning that the corresponding tuple can be removed from the cartesian product. (The two original tuples are never present in the same database version.)

The difference $T - U$ of tables $T$ and $U$ is computed as in the non-versioned case by finding pairs of tuples in $T$ and $U$ which are equal, except for the visibilities. Instead of removing the corresponding tuple from the output table, the visibility of that tuple is set to the difference of the visibilities of the $T$ and $U$ tuples. If those visibilities are represented by boolean expressions $S_T$ and $S_U$, the visibility for the
output tuple is represented by $S_{T-U} = S_T \land \neg S_U$. As for the cartesian product, in
the general case we cannot determine satisfiability for this expression. Thus, the
same problem with unnecessarily large tables is present, but in a milder form since
the size of $T - U$ is bounded by the size of $T$.

5.6.4 Binary data models

Binary data models are by many considered to be the archetypical graph data mod-
els. They can be seen as dual to the relational model, which is the basic table (or
value) oriented model [TL82, chapter 9]. For this reason we will contrast the appli-
cation of change-oriented versioning of the relational model (previous section) with
the application to binary data models, before proceeding to the Entity-Relationship
and object-oriented models, which combine features of these two main groups.

Summary of the semantic binary data model

Since binary data models are not as well known as the relational model, we will
summarize the features of a typical binary data model, the semantic binary data
model (SBDM) [Abr74].

Binary data models do not have attributes as a separate concept. Entities are atomic
(“point-like”), and relationships are simple, binary relationships without attributes.

A schema in SBDM consists of a type graph built of:

- Nodes, called categories and representing entity types, and
- arcs, called binary relations and representing a binary relationship type.

Relations have cardinality constrains, specified as a maximum and minimum card-
nality for each direction.

A database instance consists of instances of categories, called objects, and of relations,
called connections.

Objects can be either abstract or concrete. Abstract objects are considered to always
exist, regardless of whether they are referenced in the database, and are never in-
stantiated. Examples of abstract objects are values of the typical attribute domains
in other data models, such as integers, strings etc. Concrete objects are all other
objects, which can be instantiated and deleted.

Each object has a unique internal name (surrogate), which is not directly accessible
to the user. The user may specify an external synonym (which must be unique)
for an internal name. The synonym is used to identify the object in accesses. It is
also possible to create temporary synonyms (“local variables”). An object does not
need to have a synonym at all (and in that case it can only be identified through its
relationships).
"Attributes" must be represented through relationships: The domain (type) of the attribute is modelled as a category, and the attribute itself as a binary relation between the attributed category and this category.

**Versioning in the semantic binary data model**

In order to version the semantic binary data model, we attach visibilities to the connections between objects. A connection is visible for all database versions where that particular pair of objects are related by this relation type. Note that this makes all cardinalities many-to-many in the versioned database. The cardinality constraints specify the properties of any single version of the database.

In addition, if one considers the mere presence of an object significant (even if it has no connection), each object can also be given a visibility. Note that the internal name, i.e. the identity of an object is considered to be global across all versions: If objects in two versions of the database have the same internal name, they are the same object. Since objects are atomic and each has an identity separate from its relationships, it is not meaningful to talk about versioning of objects, per se.

Synonyms can be considered a relation between the (abstract) category of identifiers and objects, and versioned in the same way as other relations are by attaching visibilities to the connections.

It is interesting to note that the concept of version group (i.e. the equivalence class of all components which are versions of each other) do not enter at all: Connections have no identity, so it is not possible to identify two connections as versions of each other. Objects do have identities, but are not versioned. Thus, this demonstrates what we discussed in section 3.1.3 and 5.6.1, that it is possible to use the change-oriented model in cases where traditional version-oriented versioning is very difficult to apply.

**5.6.5 Entity-Relationship model**

The Entity-Relationship model [Che76] is a more refined graph data model, but in many crucial respects it is similar to the semantic binary data model which we discussed in the last section. The most significant difference is the presence of attributes, on both entities and relationships. Other differences, like the possibility of N-ary relationships (instead of just binary ones) and, in some versions, of subtyping, has only marginal effect on change-oriented versioning.

Due to the similarity, relationships are versioned just as in the binary models, by attaching visibilities to the individual relationships. The discussion in the previous section is directly applicable to the E-R model in this respect.

There are two ways to handle versioning of attributes. One is inspired from the versioning of the relational model, and treats entities somewhat like the RM/T model [Cod79]. The other treats attributes as merely a shorthand for the way attributes are implemented in the binary data models, namely through relationships to entities.
representing the attribute values.

Attributes as relationships

If we take a purist’s view of graph data models, we will consider attributes as relationships, the way we have to handle attributes in binary data models. Each attribute instance is considered to be a relationship between the entity and some object in the domain of the attribute.

For the data manipulation language, the consequences are:

- Attributes are normally retrieved one by one.
- For operations on multiple versions, the attribute is selected first (yielding e.g. an iterator on the multiple versions of the attribute) and then the version of the attribute is selected.

A record-oriented view of entities

In the extended relational model RM/T, the attributes of entities are represented through so-called P-relations (“P” for “property”), one for each entity type. A P-relation has an entity identifier as the primary key, and is otherwise just like an ordinary table in the relational model. If we choose to represent the attributes of entities by P-relations (and attributes of relationships in essentially the same way), it is natural to version attributes in the same way as we versioned the relational model.

We attach a visibility to each tuple in the P-relations. The P-relations can of course be normalized, in the same way as we discussed in section 5.6.3. (There may be an arbitrary number of P-relations containing properties for a single entity type.)

With these conventions, it is possible to talk about version groups. The version group consists of all fragments from the various P-relations with the same entity identifier as their key. It is even possible to say that the entity “consists” of these fragments. We work within a graph oriented data model, where the independent identity of entities is a crucial feature. Therefore, we will still maintain the convention that the entity is atomic and that attributes are associated with, rather than components of, the entity.

If we choose this approach, the consequences will be seen in the data manipulation language:

- It is natural to retrieve all attributes of a given entity as a record, in a single operation.
- When operating on multiple versions, attributes are retrieved by selecting a version first and then selecting the attribute.

The consistency criteria among the P-relations will be discussed in section 7.3.3.
5.6.6 Object-oriented models

The exact characteristics of "object-oriented data models" are a bit diffuse. Some of the more important features typically associated with object-oriented data models are:

- Object identity.
- Relations through pointers.
- Complex objects.
- Subtyping.
- Bundling of operations with data.

Instead of presenting a specific mechanism for change-oriented versioning of a particular object-oriented model, we will discuss how these features interact with versioning.

In object-oriented models, there is a tendency to regard attribute values ("instance variables") to be very strongly bound to and part of the object. (This in contrast to the binary models and partly E-R where attributes rather are associated with the entity.) Together with the strong object identity (which is always present in an object-oriented model), this makes it natural to have a notion of version group and talk about objects in different database versions as versions of each other. As we discussed in section 3.1.3, complex objects also fit naturally with version groups. On the other hand, the object identities themselves are not subject to versioning.

Pointers are really unidirectional relationships. That is, we can version pointers in the same way as we versioned relationships above. Alternatively, pointer attributes can be versioned in the same way as ordinary attributes. In practice, there are no real differences between those approaches.

The primary effect of subtyping is that different versions of the same object may have different types. This is not a conceptual problem, but does come more or less naturally with different implementation strategies. When for instance a RM/T-like strategy is chosen, with different relation tables for the different groups of attributes (properties), subtyping comes essentially for free. It is more common in object-oriented systems to store all properties for a single object physically adjacent. When in addition all versions for a single object are stored in an interleaved fashion, even type tags may have to be versioned.

As long as we consider a stable schema, (single version) operations are little affected by versioning. However, object-oriented models do typically have class objects (i.e. type descriptor objects). As soon as one considers versioning of those, things get fairly colorful. See [BKKK87] or [SZ86] for fairly comprehensive treatment of versioned types.
Chapter 6

The EPOS Database

6.1 Data Model

The data model of the EPOS database is a variant of the Entity-Relationship model [Che76], extended with object-oriented concepts, notably subtyping and complex objects. Versioning is added as an integral part of the data model, and we use the change-oriented versioning model as described in section 5.2.

6.1.1 Entities

Entities are objects with a distinct identity and existence in the database, independent of other objects.

An entity type is intensionally characterized by:

- Its name.
- An ordered list of zero or more supertypes, which the type inherits properties (i.e. attributes) from.
- Zero or more attribute definitions, in addition to those inherited from supertypes.

An entity class is the extension of an entity type, i.e. the set of instances of that type.

Each entity type may have one or more keys. Each key consists of a sequence of attributes, and allows associative retrieval. The keys must be unique within an entity class. Keys are not immutable: The same entity may well have different keys in different versions of the database.

In addition, any entity can be directly retrieved by specifying its entity identifier (surrogate). Note that the entity identifier is not an attribute, and is better thought of as a logical pointer to the entity. The identifier is immutable, and entities with the same identifier in different versions are considered to be versions of each other.
An instance of a type with supertypes is considered a collection of subobjects. There is a subobject for the type itself and one for each supertype with attributes. Each subobject contains the attributes directly defined for the corresponding type. The consequence of this is that the object may contain multiple attribute instances with the same name in different subobjects. Which one to select is determined by the qualification of the attribute access. The qualification concept is similar to the one used in Simula [DMN70] (and implicitly in C++ [Str86]). When the specified attribute is not present in the current qualifying type, the supertypes are searched in a breadth first fashion and in the order of their association with the subtype, until the first attribute with the same name is found. The attribute reference is then bound to this occurrence.

### 6.1.2 Relations

Relations express dependencies or other connections between entities. Our data model only allows binary relations.

A note on the vocabulary: The word *relationship* is used to denote an instance, that is a single connection between two entities. The word *relation* corresponds to the mathematical concept, i.e. the set of all relationships (instances) of a given relationship type seen as a whole. The relation is the class of the relationship instances, in the same sense as for entity classes. Operations on the relation as a whole (e.g. creating a relationship, asking whether particular entities are related) can be thought of as class operations in Smalltalk and other object-oriented languages.

A primary difference between entity instances and relationships is that relationships do not possess any independent identity. A relationship is identified through the pair of entity identifiers of the entities it does connect. The relationship instance is existence dependent on those—i.e. cannot exist without them. Furthermore, it has no meaning to ask whether relationship instances in different versions are the "same".

A relation may also have a key. The key consists of at least the identifiers of the related entities, and, if there can exist more than one relationship instances between a given pair of entities, one or more of the relationship attributes.

A relationship type is characterized by:

- Its name.
- A pair of roles. Each role is characterized by a name, an entity type and a cardinality.
- An ordered list of zero or more supertypes, from which the type inherits properties (i.e. attributes).
- Zero or more attribute definitions.
- Whether it is a composition relationship (see below).
Attributes and inheritance of attributes are handled identically in entities and relationships.

A role cardinality consists of a minimum and maximum cardinality. Cardinality values are 0, 1 and \( N \) (infinite). The minimum cardinality may be 0 or 1, while the maximum cardinality may be 1 or \( N \). The cardinality expresses, for one entity in the other role, how many different entities may occur in this role.

A relationship subtype inherits the roles from its parent types. The role may be inherited unchanged, or refined by replacing the entity type with a subtype, or reducing the cardinality range. There is no operation to modify relationships, and therefore all operations which are defined for instances of a relationship type are also defined for instances of subtypes.

### 6.1.3 Complex objects

Any relation may be characterized as a composition relation, and one of its roles is then designated as the component role. Viewed from a single relationship, the object related via the component role is called the subobject, and the object related via the non-component role is called the superobject. A composition relationship is not necessarily an existence dependency, only when the minimum cardinality of its non-component role is non-zero, indicating that any object of the component type must be related to some (one or more) superobject.

Any entity instance that occurs as the superobject in a composition relationship is called a complex object. Copying and deletion of complex objects affects all components recursively (i.e. "deep copying").

We allow composition relations with maximum cardinality \( N \) for the non-component role. Also, the same entity type may be in the component role for more than one composition relationship. This means that one component object may be a component of more than one superior object.

For maximum efficiency, this flexibility should not be utilized. A composition relationship should be \( 1 : N \), and no entity type a component in more than one composition relationship. This allows the DBMS to physically cluster the objects and implement copying and deletion as block copy/delete.

### 6.1.4 Attributes

An attribute is characterized by:

- \textit{name}
- \textit{domain}
- \textit{cardinality}: Whether it is single- or multivalued.
The attribute domains include: Scalar types like integer, character, boolean, user defined enumeration types and subranges of them, floating (real) and string. Values of these domains are considered to be atomic.

Long fields are generally treated as attributes, but have a different set of operations. While values of ordinary attributes are never shared among different attribute instances, long field values may be shared, and are not considered atomic.

There are no pointer attributes! All connections between entities are expressed through relations.

6.1.5 Versioning

Versioning is achieved by attaching visibilities to attribute instances and relationships. In the versioned database, all attributes are multivalued and all relationship roles have maximum cardinality "many". The cardinality constraints in the schema describes the properties of a single version of the database, which results when visibilities are bound and all attribute instances and relationships with visibility False are hidden. Entities also have an attached visibility, since a given entity need not be present in all versions of the database, and its mere presence may be significant. However, since entities are atomic they are not really versioned in themselves, only their attributes and relationships are.

Long fields are versioned as text files (see section 5.6.2). While ordinary attributes become multi-valued (i.e. sets of non-versioned primitive values) when multiple versions are considered, long field attributes are always single-valued and that single value is itself versioned.

See section 5.6.4 and the following sections for a more comprehensive discussion of change-oriented versioning of semantic, entity-relationship or object-oriented data models.

6.1.6 Operations

The database system presently offers only generic operations. We may later add user-defined method and messages as in behaviorally object-oriented systems.

The operations are implemented as library functions. The database is seen as an abstract data type, and database objects cannot be manipulated except by these functions. In particular, database objects have no main memory counterparts, directly accessible to the application. The operations are implemented by direct calls to the underlying database system, with no caching of objects except the implicit page buffering of the DBMS. Hence, updates of database objects can be assumed to be reflected directly in the persistent database.

**Symbolic names** Entity types, attributes and relations have symbolic names. For operations requiring a symbolic name to be specified, there are two variants: The
generic operation (returning a union of all possible types, in the case of attributes):

\[ \text{OP}(\text{name}, \ldots) \]

or the schema-specific one (returning the declared type for attributes):

\[ \text{OP}_\text{name}(\ldots) \]

In the first case, name is a runtime representation for the symbolic name, in the form of a small integer. These values can be defined in two ways:

- As constants \texttt{(#define)} in a schema-specific include file.
- By using a function supplied by the database for mapping string names to integer names.

**Entity operations**  In all operations, entities are identified through their object identifier. Attributes of entities are accessed one by one, instead of retrieving entities as records.

- Create entity
- Copy entity
- Update attribute value
- Get attribute value
- Delete entity
- Refine entity to one of the subtypes of its type
- Ask for iterator on a subset of the instances, based on e.g.
  - connection via specific relationships
  - value of entity attributes

**Relation/relationship operations**  The only operation directly on a relationship is the operation to get the value of the relationship. The other operations are operations on the relation as a whole. In all operations, the individual relationship is indicated by the pair of object identifiers in its roles.

- Create relationship
- Delete relationship
- Get attribute value
• Refine relationship to one of the subtypes of its type
• Ask whether two entities are connected
• Ask for iterator on a subset of the relationships, based on e.g.
  – connection to specific entity (or set of entities)
  – value of relationship attributes
  – transitive closure

**Iterators**  Multivalued attributes or other sets are not considered to be aggregate objects in their own right. They are merely collections of values or objects, often implicitly defined. To the programmer, any operations producing a set of values, for instance a `get` operation on a multivalued attribute, actually return an *iterator*.

Iterators may be considered *copies* of the sets of objects or values they represent, or direct representations of them. This affects the semantics of operations modifying the iterators, i.e. the insert and drop operations. In the first case, only the iterator itself is modified. For instance, to select a set of objects according to fairly complex criteria, first create an iterator on a larger set of objects by using a weaker criterion. Then step through the iterator, dropping any other objects that are not to be selected. In the second case, the underlying set is modified. For instance, this is how a multivalued attribute can be updated.

The operations on iterators include:

• Get current value or object
• Step iterator forward or backward
• Insert value or object (into iterator or the underlying set)
• Drop current value or object (from iterator or the underlying set)

**Long field operations**  Operations on long fields are similar to traditional file I/O-operations. (Open, close, read, write, seek, truncate, possibly open range and delete range.) To operate on a long field, the attribute `get` operation is used, which returns a descriptor for the field. (This is analogous to opening a file.) The descriptor is used as an open file descriptor in the above calls.

### 6.1.7 Methods

In principle, we would have liked to have a proper operationally object-oriented database where messages and methods are integrated with the type system. However, we consider this as a too ambitious project for the time being.

Methods, integrated with the type system, would require a new programming language (or at least implementation of that). Methods would have to be associated
with data and executed in the context of the DBMS, activated by remote procedure call or an equivalent mechanism from the application context.

Note that we do not rule out a solution with a mapping between database types and application types, where objects are converted appropriately when they are transported across the DBMS/application boundary. In this case, the application can be written in an existing object-oriented language, and methods can be associated with the application types and executed in the context of the application process.

Since we have no language for expressing operations performed in the context of the DBMS, we cannot support general triggers. (With trigger here, we mean a piece of code to be executed by the DBMS whenever a certain condition occurs in the database.) Our notifier mechanism is more restricted: Notifiers are activated by one of a handful of special conditions, and the only operation directly performed is that a signal or message is sent to the activity manager on behalf of a transaction or configuration.

### 6.1.8 Type refining

Type refining is a restricted form for subtyping. From the application point of view, a refined type behaves like a subtype. However, refined types can be created and deleted dynamically, without change in the database schema proper. Type descriptors (class objects) for refined types do exist as runtime objects, and can be examined and modified by the application.

Both entity types and relations can be refined.

- Relations can be refined by restricting the types of roles or domains of attributes to subtypes, or by restricting the range of the cardinalities of the roles.
- Entities can be refined by restricting the domains of attributes.

Instances of refined types share the database representation of their parent types. Hence, no support for dynamic schema conversions are required to implement type refining.

### 6.2 Transactions

#### 6.2.1 General properties

The EPOS database contains a long transaction facility along the lines of section 5.3. All work is performed in the context of a long transaction. Starting a long transaction creates a local workspace, and is also called checkout. Committing one returns the objects in the workspace to the parent workspace, and is correspondingly called checkin. Each transaction has an associated ambition, which gives the maximum
visibility changes done in this transaction will have when committed into the parent transaction.

The workspace or subdatabase created for a long transaction is logically disjoint from the main database, somewhat along the lines of the multidatabase facility of Damokles [DrGhLm87]. The subdatabase will contain all changed versions of objects from the main database. It is possible to have nested transactions, by making a checkout from the subdatabase. Checkin pushes the updated versions in a subdatabase up to the parent database and makes them visible in the parent transaction (and its surviving children). The main database is considered to be a transaction that never commits.

6.2.2 Copy-in and copy-out

We are mainly interested in using existing tools designed to work within an ordinary file store. To be able to use these tools, we define a mapping (in both directions) between database structures and directory trees with files in the file system. From within a transaction, a definite version can be identified by specifying a choice within the ambition, and parts of the database structure can then be copied out as a directory tree, the workspace. As long as some workspace exists, it is regarded as the master copy, and the changes made to it is propagated back to the database through the copy-in operation. Copy-in may be performed arbitrarily often, either manually activated or activated by a tool.

Some operations may not be allowed in the workspace, because it would be difficult to propagate the effect back to the database, or because the user uses tools which depend on certain aspects of the workspace and the database being in synchrony. Preferably, copy-out should set access codes on the files and directories reflecting that. Some operations, for instance modifications on the overall composition structure, should be performed in synchrony in the workspace and in the database, by special tools instead of by direct modification in the workspace. We do not rule out that a non-cooperating user or tool could make arbitrary modification in the workspace, but in that case copy-in may fail.

The workspace directly associated with a transaction will normally contain files for one or a few subsystems. In order to perform processing (for instance compilation), the activity may need access to larger parts of the system, even though it is not interested in (nor authorized to) change them. Such subsystems could correspond to, for instance, shared libraries or higher level driver modules in a program system. Such subsystems are normally much less affected by changes, and there is a high probability for each of them that it exists in the same version over a fairly large range of choices. Therefore, the system will make read-only “workspaces” corresponding to those subsystems and share them between different transactions, by means of links, logical names or another suitable mechanism.

The opportunities for sharing could be found fairly simply. A possible mechanism is to associate with each subsystem the set of options which do affect that subsystem (or its complement, depending on which set turns out to be largest). Any time a
workspace with a specific choice is to be created, and the copy-out tool examines a subsystem to decide whether it could be shared read-only, the options in the choice which do not affect that subsystem are removed. The resulting choice is used to search for an existing copy of the correct version of that subsystem, and if none is found, is used as the search key for the version that is created. The sets themselves are easily maintained by propagating information from the leaves toward the root of each subsystem.

We also note that it could — with certain restrictions — be possible to use a copy-on-write scheme, by creating an initial workspace with links to instead of physical copies of the files. This requires cooperation from the tools (or special semantics of links). The GNU Emacs editor is for instance configurable with respect to what the semantics of writing into a linked file is: Overwrite, or make a copy.

### 6.2.3 Low-level change propagation

Our high-level change propagation mechanism is based on the notification facility, which has the granularity of whole transactions. For recompilations within the workspace, this granularity is too large, unless transactions are made as small as a single editor session. On the other hand, within the workspace one (mostly) works on a single version, so a Make-like facility is sufficient. Except for possible language-dependent smart recompilation, there is little to gain by building our own tool for this. The pragmatic solution is simply to generate a “makefile” from the dependencies registered in the database at checkout time, and use the standard Make utility directly.

The tools are normally activated from the makefile, which is produced as a side-effect of the copy-out operation. Therefore, this approach gives the copy-out tool full control of the layout of the workspace, connections to shared libraries, tool versions and options etc.!

### 6.3 Handling Derived Objects

Derived objects can be handled in the general framework of the change-oriented model. “Derived objects” such as object files etc. in the traditional approaches are really versioned attribute values of particular entities (derived entities) in our database. For derived values, the database is regarded as a cache. Derived values are placed in the cache when a workspace containing derived objects is copied in. Which derived objects to cache (and for how long) is potentially subject to a user-specified strategy.

When an application asks for a particular version of a derived value (i.e. asks for that value within the context of a particular choice), the database will return that version or an equivalent one if one exists. If no suitable version exists, the database will either return an error or initiate a re-derivation, depending on what behavior the application did specify.
Derived entities have *Make or build dependencies*, which are relationships from a derived object to all objects which are required before it can be (re-)derived (its "dependees"). The database knows about derived entities and build dependencies, and assumes that all relevant build dependencies a derived object has are represented in the database. The fundamental assumption is that for all choices where the versions of the dependees stay the same, the version of the derived value is also going to be the same.

When a derived value is stored in the database, the visibility of this particular value must be determined. A sophisticated deriver tool may be able to determine a larger visibility than the database can do according to the rule above, because even if the dependees change, the new and old versions may be equivalent in the particular context the object is used. (Changed comments or unused declarations is a simple, but important example of this.) Such tools can specify an explicit visibility to use. If that is the case, and the specified visibility contains non-stable choices, it must also specify a tool the database can use later to verify that the version is still up-to-date. Otherwise, the database determines the visibility.

The visibility of a derived value that is stored in the database could include non-stable choices. A derived value which was stored with a visibility that is stable in its entirety will itself also be stable (both the value and the visibility), and can be marked as such. If a derived value is requested, for a choice which was not stable when the value was stored, the database has to verify that the value is still up-to-date. This is done by comparing time stamps, just as traditional tools like Make does. However, timestamps are versioned attributes in our model, and tells the last time this particular version was modified. If the visibility for the value was supplied by the tool, instead of computed by the database itself, the database does not immediately conclude that the value must be rederived if the timestamp of one of the dependees is newer. Instead it invokes the tool that was associated with the value, to check whether the changes in the dependees did affect it.

The consequence of this is that derived entities and dependency relations have special significance for the DBMS. We will have to supply built-in derived entities and dependency relations with fixed (but, through subtyping, refinable) semantics. Alternatively, we could have derivation and dependency properties which can be attached to entities and relations, in the same way as relations may be given the composition property.

### 6.4 Interfaces to the EPOS DB

#### 6.4.1 Notifications to the activity manager

The notification service was described in general terms in section 5.3.3. The activity manager is receiving the notifications on behalf of the transactions, and dispatching them to the destination transaction. The AM implements per project or per transaction policies for notification handling, and may for instance filter or queue notifications before they are sent to the destination transactions.
The activity manager can also be triggered from the DBMS when some activity requests a non-existent derived object version from the database. The activity manager should in this case initiate an activity to re-derive this object.

There are two possibilities for how notification messages are sent. Which one to use depends on whether the activity manager is implemented a separate process or a subroutine library linked with the main database manager process. In the first case, a notification is implemented as an IPC (Interprocess Communication) message, possibly a remote procedure call. In the second case, the notification is simply a normal procedure call (possibly a callback) from the database subsystem to the activity manager subsystem.

In both cases the logical contents of the notification are the same. All notifications include an identification of the configuration description or transaction this notification is on behalf of. This identification is used by the activity manager to multiplex the notification to the correct destination. The rest of the notification depends on its type:

1. This configuration description needs reevaluating. Associated information: List of options which will change state, list of affected objects.

2. Another activity started a transaction with overlapping ambition. Associated information: Specification of the overlapping region in choice and object space, identification of the other activity.

3. Another activity committed a transaction with overlapping ambition. Associated information: Specification of the overlapping region in choice and object space, identification of the other activity.

4. A derived object in this configuration needs re-derivation. Associated information: Identification of the object.

The information associated with a notifier needs not necessarily be directly included in the notification message. It could also be produced lazily, by having the process receiving the notification explicitly asking for it. This would have two purposes:

- Avoid unnecessarily large notification messages
- Avoid deriving information which is not called for

For case 1 above, the list of changed options would have to be produced in any case. But as soon as the first affected object in the configuration was found, the notifier could be sent. Then the recipient would only ask which objects were actually affected in the case where it does need this information.

For case 2 and 3 above, the information about the overlapping region could be delivered lazily, but (most of) it would be available before the notification was sent anyway.
Chapter 7

The EPOS Prototype

7.1 Overview of the Initial EPOSDB Prototype

7.1.1 Description

The first version EPOS prototype database (EPOSDB) will function as a versioned
archive for program products in much the same way as tools like for instance SCCS
or RCS. However, the EPOSDB will be versioned according to the change-oriented
model described above, and it will also contain and integrate information about tools,
derivations and configuration building. This is roughly the same information which
is stored in the “makefiles” when the conventional Unix tool Make is used. However,
we will integrate this information much more tightly with versioning, allowing for
instance caching of derived objects to accelerate rebuilding of “nearly identical”
configurations.

Each EPOSDB instance contains the information about a single program product.
This does not necessarily imply a single executable program, but for instance a
family of interacting programs sharing common libraries etc.

The information contained in an EPOSDB is:

- Primary and derived components\(^1\) themselves—essentially versioned files, with
  the same granularity as files in conventional development environment.
- Structuring information, grouping the components into subsystems.
- Information about component types and applicable tools.
- Derivation steps required to produce the product or parts of it.

Compared to a traditional Unix development environment, EPOSDB offers func-
tionality comparable to the combined functionality of SCCS or RCS and Make, but
with stronger integration of versioning and rederivation.

\(^1\)Avoid the word “module” on purpose: Without further qualification, that term is ambiguous
and could mean either a component of some unspecified granularity—a file or a set of files, or a
specific language construct in languages like Modula.
The database is not manipulated directly with normal tools. Instead a version of the product is copied out of the database in the form of ordinary files in a workspace. The product structure is reflected in a Unix directory structure, while the derivation information is used to produce a Makefile. The native Unix Make is used to drive the actual rederivations in each single-version workspace. In later versions, Make will be replaced by the EPOS Process Management component as a derivation driver.

A primary difference with respect to SCCS and RCS is that derived components (object files) are also archived. Each derived component which is copied-in is analyzed by EPOSDB to determine for what range of product versions this component version is valid. If the archive contains the appropriate version of a derived component when a product version is copied out, the derived component is also copied out and needs no rederivation. When copying out files, the update timestamps of the files are set according to the timestamps stored in the database, and vice versa for copying in.²

### 7.1.2 Functionality offered

The following main functions are offered by the initial EPOSDB prototype, implemented as Unix commands (executable programs or shell scripts):

- Create database.
- Create option.
- Automatic derivation graph building ("planning"), based on tool descriptions.
- Copy out version of database and create workspace, given ambition and choice (specified as textual, boolean expressions).
- Copy back workspace into database.
- Tools to manage product structure etc. in the database.

The first prototype does not contain any higher-level functions for management of options (such as higher level version descriptions). Such functions are considered, and are undergoing design, and later, we plan to add them on top of the prototype as described here. What is implemented in this phase is just the basic platform for change-oriented versioning, namely version selection and checkout, checkin, transaction contexts with ambitions. Stability and validities are somewhat more critical that version descriptions, and will be added to the prototype as soon as the basic functionality is implemented.

There is a library function interface to the database (C and Prolog), but it will be implemented with the main purpose of supporting the above-mentioned tools. There is no direct interface to the database which can be used by existing tools.

¹Not all operating systems have the capability to set update timestamps, as Unix has. In this case, some approximation, like copying out files in the order of their database timestamps, would allow timestamp-based derivation tools like Make to work correctly.

115
Figure 7.1: Product archive with workspaces
Only a simplified transaction model will be implemented. In this model, there are no subdatabases, except for copied-out workspaces, and overlapping transactions are not supported in any major way (although permitted).

### 7.1.3 Database bridges

Copying out files into workspaces and making sure these workspaces conform to normal Unix conventions allows us to use the standard tools like Make, compilers, loader, archive builder etc. However, EPOSDB must be open enough to allow some degree of integration with certain other tools, e.g. NEXPERT[Nex88] or Software through Pictures[WP87]. We plan to support those with tools to copy out data in formats compatible with these packages. Although these tools will be non-generic, specific to each data format, they will be very similar conceptually to the main “file workspace manager” application described above. Many of the same mechanisms for handling copyout, copyin and change propagation will apply.

In particular, copyout of structural information into a relational database (RDB) is deemed important, considering the large number of potentially useful tools which have RDB interfaces. Alternatively, if the kernel of EPOSDB itself is based on a RDB, we could permit tools with an existing RDB interface direct access, but there are serious problems with that:

- The EPOS data model introduces a number of integrity constraints which cannot be expressed in SQL, and thus must be maintained implicitly by the applications accessing the underlying database directly. (We cannot introduce our own database access libraries in existing applications.)

- Versioning can be handled transparently only for read-only applications.

- A lot of the general database “baggage” is of dubious value, or outrightly wasted: E.g. transaction management or concurrency control, which is handled by copyout/copyin in our case.

### 7.1.4 Purpose of the prototype

The purpose of the prototype activity is to try our approach—in particular basic change oriented versioning and caching of derived objects—in a “real” development. This implies that we have to build a prototype system which is robust and complete enough that it is possible to maintain a moderately complex piece of software in it—for instance the prototype itself. This in turn implies that we will have to limit ourselves to implement those parts of EPOS which are well enough understood by now.

Unless we are able get prolonged experience with a realistic development using the prototype, the prototype itself is essentially useless. Our aim must be to demonstrate that our system offers significant advantages over for instance RCS and Make!
7.2 The Design of the EPOSDB Prototype

The central database contains the real change-oriented product archive, i.e. versioned files (objects) and a versioned product structure. The files are organized in a generic subsystem structure, with properties rather like a hierarchical file system. The database also contains information about tools, potential and actual derivations. Components and subsystems have types, which determine which tools are applicable to them.

Around the database one or more workspaces exists. Each workspace is implemented as a Unix directory hierarchy. The directory structure mirrors the product structure in the database, i.e. directories corresponding to subsystems and files to components. Each workspace is associated with a long transaction in the database.

A workspace and its corresponding transaction is created by the copyout operation. A directory structure is created, and populated by files corresponding to the requested version of the components. In addition, the information about how to build derived objects is extracted in the form of makefiles. In most cases, this requires a "planning" step to be run in the database before the makefiles can be created, in order to create the derivation graph from the types of objects, applicable tools and import/export relationships.

When the owner of a workspace is satisfied that the task is finished, the workspace is copied in to the database again and deleted. Changed versions of files and directories are noted and the information in the database is updated accordingly.

The transaction model which is planned for the EPOS database calls for nested, long transactions within the database, and a fairly sophisticated concurrency control mechanism based on controlling overlapping ambitions. The full model will not be implemented in the first version of the prototype, mainly because it is impossible to implement with reasonable efficiency on top of an ordinary database system. Long transactions are simply workspaces, and concurrency control is reduced to locking the whole database during copyout or copyin. If a transaction is started up with an ambition overlapping with another transaction, both transactions are notified, but they are allowed to proceed with no further assistance from the system to reconcile their changes. I.e. the changes from the first transaction to check in will be lost, unless the transactions were reconciled manually before copyin. In the next version of the prototype database, transactions will be fully implemented according to section 7.3.

A listing of the database schema, as well as a more implementation-oriented explanation of its semantics is found in appendix B. Appendix A describes the Prolog programmer's interface to the versioned database.

7.2.1 Product structure

The components are the primitive, and as far as the product archive is concerned, atomic units in the archive. Components are essentially a generalization of files.
Subsystems are used for aggregation and naming, just as directories in the Unix file system, and do in fact share most of their properties with Unix directories. Names are relative to subsystems, there are no absolute names. A subsystem is essentially a set of (name, part) mappings (where a part is either a component or a subsystem. The same part may be a member of several subsystems, but the product graph is required to be acyclic.

We chose this structure for the following reasons:

- A hierarchical file system is an established and useful generic archive structure. ("Hierarchical" in the sense of the Unix file system, whose structure actually is a directed, acyclic graph.)
- There is a straightforward mapping between database and workspace—in both directions.
- The product archive has to be open ended, to incorporate system libraries, existing tools etc. which it is inconvenient to copy into the archive. Ideally, the directory structures in which these are stored (e.g. /usr/include, /usr/lib in a Unix system) could be "grafted" onto the product archive.

Several other objects, e.g. libraries and code files, can abstractly be considered to be collections of named objects, and we have considered whether they should be represented as subsystems or atomic components in the scheme above. Libraries and object modules differ from subsystems in the sense above in that they contain physical copies of objects rather than references to them. Also, libraries are incrementally built—that is, members are usually added or replaced individually.

7.2.2 Tools and derivations

The product archive contains information about tools and derivations which is used for three distinct purposes:

1. Recording derivation history for derived objects, to avoid rederivation unless a significant change occurred.

2. Describing how a given derived object can be created.

3. Describing tools, to which objects and under what conditions they can be applied.

(1) and (2) are strongly related, and do in fact use the same structures in the archive, namely the derivation graph, relating each derived object to all objects it was derived from (i.e. all objects which were read by the derivation tool in order to produce the derived object).

The derivation graph is produced by matching (3) against the derived objects which are to be produced. This process will be called planning, by analogy with planning
in artificial intelligence. When copying out, the Makefiles are produced from the derivation graph. Changes in tool or product structure may require replanning and subsequent rederivation of Makefiles during a transaction, however.

The contents ("value") of a derived object is considered to be an attribute, and may be undefined (i.e. before the derivation step has been run). Consequently, there is no need to take special precautions for not-yet-derived or out-of-date objects when the derivation graph is built.

A derivation step may produce more than one derived object. One special case is not currently catered for: Each of the output objects need not depend on all input objects. With the database structure as specified below, there is no way to record that. Some derivations may in fact be redundant, if only the non-significant input objects were changed.

See section 7.2.4 for examples.

7.2.3 Building the derivation graph

The purpose of "planning" in our context is to build the derivation graph, starting from a set of derived objects which are requested and working backwards towards primary (non-derived) objects.

Planning proceeds by considering each requested object and attempting to match the object to any of the described tools and its declared input and output parameters. If a matching is found, a derivation step object is created, linking the matched input and output objects and connected to that tool. The matching is done in two steps, one performed by a generic matcher in the planner and one by a tool-specific procedure:

1. Look for a tool with an output parameter of an appropriate type. Check that all input and output parameters for this tool can be satisfied with objects of the correct types and which otherwise match the criteria given in the parameter descriptions. (Currently: Only matching on the names of the objects, similar to the way Make handles implicit rules in Makefiles.)

2. Pass the matching found in the first step to a tool-specific matcher procedure. This procedure is responsible for completing the derivation step node which was initiated: Finding any additional dependencies (like #include's for the C compiler) and adding arcs for them, building the command line for the tool invocation or any other tool-specific action. It could also determine that it is not in fact possible to derive the object in this way and return failure to the planner.

Planning can itself be considered a derivation, and it is tempting to use the same mechanisms to detect a need for re-planning as for detecting normal out-of-date derived objects. The problem with this is that the result of planning is a derivation graph, i.e. a set of relationships in the database, not an object as for normal derivations. Our current data model makes it difficult for us to consider the derivation
graph an object in its own right. A complex object mechanism like in Damokles (see section 4.3.1) would make that possible.

In fact, the build dependencies for a derivation step (i.e. for regeneration of the derivation step itself) are a subset of those for the object(s) produced by the derivation step. If the inputs or attributes of a derivation step are changed, the output object will surely have to be rebuilt! Thus, an alternate way to determine that replanning is necessary is to assume that the matcher procedure adds input dependency arcs for all objects it had to examine in the process of determining the dependencies. For any out-of-date object the dependencies will have to be regarded as suspect and re-validated before a rederivation is attempted. (Intelligent editing tools might decorate the objects with information to accelerate this process.)

The latter mechanism turns out to be better than the first, perhaps more elegant and "orthogonal" one, which treats planning as a just a special case of derivation. The reason is that it permits replanning at the finest grain, a single derivation step. The first would require some mechanism for incremental rebuild if it is to make sense, otherwise the whole dependency graph would have to be rebuilt each time an input object changes.

7.2.4 Tool modelling

With the mechanisms described above, there is little deep knowledge about tools explicitly represented in the database. Most of the knowledge is embedded in the matcher procedures, which determine if a tool is applicable in a given situation and sets up the details for the tool activation. Of course, the matcher procedures may use additional information which also is recorded in the database, by tool-specific extensions to the basic database schema.

Let us consider how some of the important Unix tools can be handled in our framework. We will not get too specific in details depending on conventions about how files are organized into subsystems, usage of libraries etc. Also note that the accompanying figures are somewhat simplified with respect to the text, in order to not get too cluttered with detail.

C compiler

The compiler description (figure 7.2) specifies two input and one output parameter, an input object of type C-source with name pattern $(name).c, another input object of type cc-switches, and an output object of type relocatable and with name pattern $(name).o.

The matcher performs the following actions (see figure 7.3 and 7.4):

1. Determine the closure of #include's in the source file.

   (a) To resolve the names, follow imports links to imported subsystems and perform name lookup there. The standard C library is considered a
subsystem here (i.e. there is an imports link to /usr/include).

(b) Alternatively, #include’s could be represented by explicit links in the database.

2. Set the source-language attribute of the output object to “C”. Also, set the main-program attribute if this is a main program.

3. To allow the loader to find the correct relocatable objects or libraries corresponding to externals and #include'd files, attach this information to
the output object. The information itself should be possible to state in a language-independent format, but the derivation of it is language-specific. For C code, the matcher could rely on "implements" links from the .h-files (by convention used as interfaces) to the .c-files containing the definitions for the procedures and variables declared in the .h-files. Then a link is created from the output object to those .c-files or the relocatable objects derived from them.

4. Build the command line for the compiler invocation. In particular, insert switches from the cc-switches input, and also -I switches corresponding to imported subsystems.

Most compilers under Unix are similar and can be handled in the same way.

Loader

The loader description has an input parameter of type relocatable and pattern $(file).o, an input parameter of type ld-switches and an output parameter of type executable with pattern $(file).
1. The matcher first checks if the relocatable input has a `main-program` attribute and rejects the matching if not.

2. The list of additional required relocatable objects and libraries is built by extracting the information the compilers (or the matcher for the compiler) have attached to each relocatable object and performing a closure.

3. The command line for the loader is built, taking into consideration the `source-language` attribute of the main relocatable module.

### 7.3 The Prototype Database

The prototype EPOS database will be implemented on an underlying relational system, either a proper relational database system, or, most likely, ISAM files. Entities and relationships in the EPOS data model are mapped onto tables in a fairly conventional way. Versioning is handled primarily on the table level, by attaching visibility fields to the table and filtering based on the visibilities.

#### 7.3.1 Versioning and transaction management for tables

We have several choices with respect to how our database is mapped onto tables:

1. A private set of tables for each long transaction. This gives excellent properties with respect to distribution and isolation, but the entire database (or relevant part of it) has to be copied on checkout. It follows that the user will see the shared data in the database as of the time of the checkout. Changes from committing sibling transactions are not visible.

2. One set of tables for each long transaction, with only the modifications with respect to the parent transaction stored. To find a particular datum, the tables making up a “logical table” are searched, in prioritized sequence, depending on which transaction the access is in context of. Since each long transaction only writes into one physical table out of all tables making up a logical table, no locking is really needed except during the actual checkin operations. (Alternatively, table locking is sufficient).

3. One set of tables for all transactions, tuples are marked with flags telling which transaction they belong to. This solution makes it impossible to use table locking, and could be inefficient because of a potentially high number of tuple locks.

We choose alternative 2, because this combines some of the properties of the two other solutions.

The mechanism we describe can be seen as implementing a long transaction and versioning facility within the relational model. A relational database without long
transactions has a set of physical tables. The corresponding database with long transactions has a set of the same number of logical tables with the same definitions. Each logical table is implemented by a set of physical tables, one for each long transaction. The local tables of a given transaction are the physical tables (the cross section across all logical tables) owned by this transaction.

The virtual contents of the logical table, as seen from a given long transaction, is the set of tuples in the union of the local table for this transaction and the corresponding ones in all its parents. However, a tuple in one of the local tables hides all tuples with the same primary key in the local tables of all the parents of that transaction.

Each write or update of a tuple in the logical table causes this tuple to be added to the local table (unless it already resided there). The effect is that this transaction (and its children) will see the new value, while all other transactions with overlapping ambitions will continue to see the previous value until this transaction commits.

A local deletion causes a specially flagged tuple to be inserted in the table, with the semantics that if such a tuple is found for a given key, the table behaves just as if no tuple with that key is in it at all.

Unfortunately, the construction of the logical table as a SQL view is not possible, chiefly because SELECT conditions may only take into account a single tuple.

**Versioning**

All tables include a visibility column with the name S, whose domain is an integer, the internal name of a visibility predicate. The special table PREDICATES maps the predicate names into the corresponding boolean expressions. It consists of a S column and another column which is a representation of the predicate. In order to perform version filtering and get a definite version of the database, all predicates in the table PREDICATES are evaluated, and a new table S is created with just a S column and containing only the visibilities that evaluated to True.

It is possible to implement version filtering as a SQL view, for read-only access to the database. Each logical table $T(A_1, \ldots, A_n)$ is implemented by the following SQL statements:

```sql
create table vT ( S, A_1, \ldots, A_n );

create view T as
  select A_1, \ldots, A_n from vT
  where S in (select S from S);
```

The view $T$ consists of all tuples from $vT$ with a $V$ attribute which is present in the $V$ table (i.e. evaluates to True). However, database update is more complex than what can be handled by the SQL view mechanism.

Each subtransaction has an ambition which is a subset of the ambition of the parent transaction. The PREDICATES table is not treated as a logical table, instead there
are separate tables for each transaction. Initially, the PREDICATES table is created as a copy of the parent’s, but the predicates are modified by inserting values for all options defined in the ambition and simplifying. All new visibilities \( S \) created in this transaction are also effectively \( S' = S \land A \). Since predicates are not modified in-place, all visibilities that were in the initial PREDICATE table are known to be the same in this transaction and its parents when the time is coming for commit. For the other visibilities, the predicate is modified by “anding” with the ambition, to get the effective visibility with respect to the parent transaction.

**Transactions**

A new transaction is started by creating empty local tables for it. It is committed (checked in) by copying the contents of its local tables into the local tables of the parent transaction and deleting the local tables of the child transaction afterwards. The visibilities in the child tables are updated to \( S' = S \land A \). The checkin is run as a transaction in the underlying relational DBMS. Either checkin completes and the child tables are deleted, or checkin fails and the parent and child tables are left untouched such that checkin may be retried. The failure may be due to an update performed directly in the parent transaction, or a sibling transaction that commits concurrently.

If the child’s and parent’s local tables contain tuples with the same key, the tuple in the parent table is replaced with the one from the child for all choices in the child’s ambition. This is done by comparing the visibilities. If the parent’s visibility is contained in the child’s visibility, the parent’s tuple is simply overwritten. Else, its visibility \( S \) is changed to \( S' = S \land \neg S_{\text{child}} \), where \( S_{\text{child}} \) is the visibility associated with the corresponding tuple in the child, and the child’s tuple is inserted into the parent table. The “deletion” tuples which act as placeholders for deleted tuples in the local tables are treated as a special case. If such a tuple is found in a child table, the update of the parent table proceeds as usual, except that no new tuple is inserted.

This scheme of simple replacement can be refined by introducing timestamp-based, optimistic, concurrency control.

### 7.3.2 Mapping the EPOS data model on tables

After defining logical tables, embedding versioning and transaction control, we can implement the EPOS data model in terms of this database. The word “table” below refers to a logical table in this logical database. Note that we could just as well implement the EPOS data model directly on top of physical tables, but then without versioning and long transactions. The discussion below is just as applicable in that case, and then the word “table” must be taken to mean a physical table.
Entities

For each entity type, including each subtype, there exists an table, called a property table. The property table associated with type $T$ is denoted $P_T$. The columns of the property table are:

- Visibility
- Object identifier
- One for each immediate attribute (not inherited) of the type.

The primary key of each property table is the object identifier and the visibility, in that order. After version filtering, the visibility column vanishes, and the primary key becomes the object identifier alone. We denote a property table after version filtering as $P'_T$.

The names of the attribute columns are the attribute names. The visibility and object identifier columns have the reserved names $S$ and $O$.

Subtyping of entities

The tuples representing a specific entity are found in the table for the type of the entity itself and the tables for all its supertypes.

All objects of a given type are produced by joining the property or relation tables for the type and all its supertypes by a natural join over the $O$ columns. Note that this will retrieve all objects of any subtype of the given type also, but only the columns
(attributes) defined to belong to the given type. This is consistent with treating objects of a subtype as instances of the supertype at the same time. Also note that the type of an object is defined implicitly by which tables the tuples making up the object are found in. An object may have different types in different versions simply by having different visibilities for its tuples. To refine an object into a subtype, just add a tuple for it in the table of the subtype.

When a complete object of a specified type is to be read in, we would have to search for its subobjects in the property tables for all subtypes of that type. Our scheme can be refined by adding a “type tag” column to each property table, which tells us the immediate subtype and hence the next property table to consider. For instance, if $T_1$ and $T_2$ are subtypes of $T$, a tuple in the property table of $T$ has a tag column containing $T_1$ if it is of type $T_1$ or any of $T_1$’s subtypes.

Note that the join assembling the fragments of the objects is performed after version filtering, i.e. on a specific version. If joining before version filtering, the visibilities also have to be taken into account in the join condition: When two tuples are joined, the tuple in the join table has a visibility which is the intersection of the visibilities of the original tuples. If the intersection is empty, no tuple occurs in the join. Due to the NP-completeness of determining boolean expression satisfiability, this procedure is not practical in the general case. (See section 5.4 and 5.6.3.)

Spreading the tuples for individual objects over several tables like this does reduce performance. In return, to find all objects of a particular type, it suffices to search one table.

**Relationships**

Relationships do not have object identity. For this reason, subtyping of relationships is implemented differently. For each relation type, including each subtype, there exists a table, called a relation table. The columns of the relation table are:

- Visibility
- The roles
- One for each attribute (both immediate and inherited) of the type. (Note the difference with respect to entities.)

The primary key of each property table is the roles and the visibility, in that order. After version filtering, the visibility column vanishes, and the primary key becomes the roles alone.

The names of the attribute columns are the attribute names. The names of the role columns are the role names. The visibility column has the reserved name $V$.

To find all relationships of a given type, it is necessary to search the table for the type itself, as well the tables for all subtypes of that type.
Attributes

The attribute types in the EPOS data model are directly mapped into the corresponding field types of the underlying DBMS.

Long fields are mapped into string attributes representing file names. These attributes are given special treatment at checkout and checkin.

7.3.3 Consistency constraints and visibility update

There are several consistency constraints governing the update of visibilities and contents in the property tables. We will write $S$ is an immediate subtype of $T$ as $S \prec T$, and that $S$ is a subtype of $T$ in general as $S \prec^* T$ (i.e. the transitive closure of $\prec$). We also adopt the convention that any type which is not a subtype of any other type is implicitly a subtype of $\top$, i.e. for all types $T$, $T \prec^* \top$. Logically, we can think of the property table for $\top$ as having a single tuple, with visibility True, for each known object.

We assume for the time being that only single (tree-like) inheritance is supported. The following constraints must hold for all possible versions of a database:
• **Uniqueness of user keys** was discussed in section 5.6.3. (Here, the user key is the object identifier $O$.) A peculiar key uniqueness constraint is introduced by the possibility of multiple subtypes. Alternate subtypes of a given type are all mutually exclusive, so the same object identifier may not occur in more than one of those property tables.

• **Foreign key constraint:** An object which has a particular subtype $T_i$ also is a member of all the supertypes of $T_i$, i.e. if a tuple for an object exists in $P_{T_i}$, there must exist tuples for it in the property tables for all supertypes of $T_i$.

More precisely:

**Constraint 1** For all pairs of types $(T_i, T_j)$ such that neither $T_i \preceq T_j$ nor $T_j \preceq T_i$, $\pi_O(P_{T_i}) \cap \pi_O(P_{T_j}) = \emptyset$.

**Constraint 2** For all pairs of types $(T_i, T_j)$ such that $T_i \preceq T_j$, $P_{T_i} \subseteq P_{T_j}$.

It is possible to relax constraint 1 to a more easily checked condition:

**Constraint 3** For all pairs $(T_i, T_j), T_i \neq T_j$ of immediate subtypes of a common type $T$, $\pi_O(P_{T_i}) \cap \pi_O(P_{T_j}) = \emptyset$.

We can show that constraint 3 and 2 imply 1 as follows: Any pair of types $(T_i, T_j)$ such that $T_i \not\preceq T_j \land T_j \not\preceq T_i$ must have a common ancestor $T$, i.e. $T_i \prec^* T \land T_j \prec^* T$. But there are immediate subtypes $T_i'$ and $T_j'$ of $T$ such that $T_i' \preceq T \land T_j' \preceq T \land T_i' \preceq T_j'$, and constraint 3 guarantees that $\pi_O(P_{T_i'}) \cap \pi_O(P_{T_j'}) = \emptyset$. But it follows from constraint 2 that $\pi_O(P_{T_i}) \subseteq \pi_O(P_{T_i'}) \land \pi_O(P_{T_j}) \subseteq \pi_O(P_{T_j'})$, and consequently that $\pi_O(P_{T_i}) \cap \pi_O(P_{T_j}) = \emptyset$.

We stated the constraints in terms of the database after version filtering, because they are more intuitive that way. In order to ensure that the constraints hold for any possible version of the database, they have to be restated in terms of visibilities in the database before version filtering.

For the same $T_i, T_j$ as above, constraint 3 becomes:

**Constraint 4 (Uniqueness of keys)** For any pair of tuples $(A_i, A_j), A_i \in P_{T_i} \land A_j \in P_{T_j}$ such that they agree on the object identifier $O$, their visibilities $V_i$ and $V_j$ must be non-overlapping, i.e. $V_i \land V_j \equiv \text{False}$.

Constraint 2 is replaced by:

**Constraint 5 (Foreign keys)** For any pair of tuples $(A_i, A_j), A_i \in P_{T_i} \land A_j \in P_{T_j}$ such that they agree on the object identifier $O$, the visibility $V_i$ of $A_i$ must be contained in the visibility $V_j$ of $A_j$, i.e. $V_i \Rightarrow V_j$.
The update algorithm is called with the ambition $A$ and an object of type $T_O$ with object identifier $O$:

1. For each visibility $S$ occurring in the database, define $S_r(S) = S \land \neg A$ and $S_n(S) = S \land A$

2. Set $\text{NewVis} = A(= S_n(\text{True}))$.

3. For each type $T_{curr} \geq^* T_O$, taking supertypes first, consider its property table $P_{T_{curr}}$:
   a) Set $\text{Nochange} = \text{False}$.
   b) Examine all tuples with object identifier $O$ in $P_{T_{curr}}$ and all $P_{T_{sub}}, T_{curr} \prec T_{sup} \land T_{sub} \prec T_{sup}$:
      i. If a visible tuple was found:
         A. If its value is equal to the tuple about to be written, set $\text{Nochange} = \text{True}$.
         B. Otherwise, update its visibility $S$ to $S_r(S)$ and set $\text{NewVis} = S_n(S)$.
         C. If the tuple was in $P_{T_{sub}}, T_{curr} \neq T_{sub}$, consider tuples in all $P_{T_{sub}}, T_{sub} \prec T_{sub}$ and update each of their visibilities $S$ to $S \land \neg \text{NewVis}$.
      ii. If none of them were visible: For each of the tuples, with visibility $S$, set $\text{NewVis} = \text{NewVis} \land \neg S$.
   c) Unless this is a deletion or $\text{Nochange} = \text{True}$, insert a new tuple in the property table, with data from the corresponding subobject of the current object and visibility $\text{NewVis}$.

Figure 7.7: Database update algorithm

Checking these constraints on the visibilities without any a priori knowledge is NP-complete. (They involve determining the satisfiability of a boolean algebra expression.) However, the update algorithm in figure 7.7 avoids the explicit checking, at the cost of not ensuring that some redundant tuples remain in the database.

Note: In all cases when visibilities are updated, it is possible that:

- The new visibility is equivalent to $\text{False}$, in which case the tuple can be deleted, or
- the new visibility is the same as the old one.

For the performance of change-oriented versioning, it is important that these conditions are detected most of the time they occur. However, it is not essential for
the method that they are *reliably detected* each time, which would be impossible because of the NP-completeness of boolean satisfiability. The cost is that a number of redundant tuples may build up in the database over time.

The loops in the above algorithm range over the supertypes of the actual type of the object and the fragments of the different versions of the same object, respectively. For neither of these would the numbers be expected to increase significantly with the total size of the database, but the last would probably increase slowly with the number of transactions performed. Empirical results are required to verify these guesses, though.
Chapter 8

Evaluation and Conclusions

8.1 COV Evaluation

8.1.1 EPOSDB prototype

Unfortunately, the implementation of the EPOSDB prototype has taken significantly more time than expected—for reasons unrelated to change-oriented versioning (manpower, unfamiliar tools etc.). At the time of writing, it is still not possible to run the prototype.

However, in some preliminary work [Mun89] a tool to store text files in a change-oriented fashion was built. Here, there were fairly serious problems with the manipulation of boolean expressions. It is too early to decide whether these were due to an unsuitable tool (a design tool for Programmable Logical Arrays called Espresso [Rud85]), or is more fundamental. Some source files under control of RCS were used as testing material. Each file had from 6 to 28 revisions, and the size of the RCS “database” altogether was 260 Kbytes.

Revisions with corresponding numbers in the different files were assumed to correspond, and an option was created for each revision number. Then all revisions were checked into the change-oriented archive. Due to the sequential nature of RCS revisions, the growth of the number and complexity of the visibilities were only moderate. However, a couple of synthetic edit tasks grew the size of the file storing the visibilities by a factor of 5. We are not convinced that these edit tasks presented a realistic test case, however. For instance, the changes were uniformly distributed throughout the files. As we have argued, there is reason to believe that distinct options control changes which are largely disjoint in the file. Also, the changes done in the synthetic change jobs were extremely large: every third line was changed.

Checkout of the last version of the largest file took 16 seconds, comparable to the time spent by RCS for the same operation. Checkout of internal versions took significantly longer time, several minutes. The largest checkin took over an hour, computing 89 new visibilities. These figures are not very relevant, due to weaknesses or outright mistakes in the implementation which are possible to remedy:
• It is not necessary to handle expression evaluation by the general expression
  simplifier. Just insertion of choice followed by simple evaluation should make
  checkout significantly faster. Our estimate still remains that for reasonably
  sized files, the time to read through the file completely dominates the time for
  the initial expression manipulation.

• Optimal or near-optimal simplification is not critical for visibilities. As it was
done, the heuristic algorithm implemented by the tool we used failed for the
more complex cases. If the running time exceeded a set limit, it was simply
aborted and a simple, fast algorithm which essentially managed no simplifica-
tion at all was substituted. A more incremental algorithm or implementation
which could be aborted without throwing away the results would have been
much more economical with the CPU time.

• The expression simplifier was implemented as a separate program, running on
a separate machine to boot. Thus at least a 3-4 seconds delay was incurred
each time it was started, and the implementation made few attempts to batch
expressions to simplify.

The delta files were on the average 10% smaller than the corresponding RCS delta
file. Somewhat smaller deltas might be possible by using a smaller grain size than
whole text lines. RCS also uses line-by-line deltas and in fact the same difference
processor (UNIX diff), so it would have been a major surprise if COV and RCS
deltas were not within a few percent of each others: In both cases, text contents
dominate administrative overhead in the files. However, COV needs the separate
files to store visibilities and other predicates, these grew to rather significant sizes.
As long as only sequential revisions from the RCS files were checked in, each checkin
added one new visibility (but the visibilities themselves gradually became more com-
plex). A synthetic edit task, modifying every third line of a version, did on the other
hand create 89 new visibilities.

Clearly, if change-oriented versioning is to be a practical alternative, we need to
handle boolean expressions in a more efficient manner. With the prototype running,
we expect to get a better feeling of exactly where the real bottlenecks are and start
working to remove them.

8.1.2 Complexity issues

The complexity issue was discussed from a technical point of view in section 5.4.
Clearly, this is one of the most crucial issues to be resolved if COV is to be a
practical methods. We found that it is possible to keep the complexity mostly under
control by making restrictions on the form of the ambitions. Of the issues involved
in implementing basic change-oriented versioning, only the question of the growth in
the number of visibilities remain unresolved. Since it is so dependent on the actual
usage patterns, only experimenting with the prototype will be able to resolve that
question.

Some other issues involved in more advanced applications (in general, multi-version
analysis) depend on reasonable heuristics for determining satisfiability of a boolean
expression. We have been using a tool for boolean simplification which was originally developed for construction of VLSI Programmable Logic Arrays. ("Espresso" from the Berkeley VLSI tools suite [Rud85].) This tool relies on heuristic methods which may not be particularly appropriate to our application area. An interesting topic for further research is whether it is possible to devise more efficient heuristics for our simplification problems than those used in the PLA-simplification algorithms.

At this point, we have no experiences as to what form the expressions will take in typical use of COV. It is conceivable that experimentation may reveal patterns which we did not anticipate in our theoretical analysis, which could form a basis for heuristic methods. For instance, expressions may originally have shared subexpressions which are lost if a general expression simplifier is turned loose on the expressions. Of course, we cannot rely on this to save us from otherwise hopeless complexity.

For those complexity issues which are critically dependent on actual usage patterns, for instance the number and complexity of visibilities, it is not clear that potential complexity in itself is an argument against COV. As mentioned in section 5.4, if orthogonal changes are cleanly modularized in the software system, the visibilities never get many and complex. Thus, complex visibilities may actually be a symptom that the flexibility of COV is overused. Similar arguments might be used about other cases also. After all, if NP-completeness rears its ugly head, how could one expect human users to keep an overview?

The issue is then not so much "Is COV too complex?", as "Does COV permit too much complexity to be expressed?" Traditional versioning tools constrain complexity simply by making it inconvenient or impossible to represent very complex version structures. For instance, version merges or parallel versions is usually so difficult to work with that it is not done, even when it from a conceptual point of view is most natural. If only sequential revisions are implemented in COV, nothing is going to be very complex there either.

The most important question is actually: **Is it possible to constrain the development process enough to avoid stumbling into unmanageable complex product structures, without loosing the potential advantages of COV?**

### 8.1.3 COV advantages

What advantages do then COV have?

- It offers simpler development of parallel, orthogonal versions where changes done in different contexts must be merged together. The merging in itself is not any simpler, but the framework around it is more convenient to work in, in our view. Through the ambition, it is at least possible to propagate changes more precisely into those versions where they belong.

- COV automatically selects corresponding versions for each system component. A given choice basically selects a bound configuration, the version for each component is uniquely determined. We have already pointed out that this
does in no way mean that the same component version may not occurs for se-
veral choices. Thus orthogonal combination of component versions is implicitly
taken care of.

• In our view, COV is more widely applicable to versioning of databases. VOV
assumes that it is possible and meaningful to form version groups. COV does
not make any such assumptions, and does for instance easily permit versioning
of relationships.

• Options and ambitions connect versioning to the development process in a
natural way. Options correspond to user-visible functional features of the
product, which may even be meaningful for personnel outside the development
organization. Ambitions capture the intention of a particular development
task, much more precisely than for instance log entries in SCCS/RCS files.

8.2 Related Work

8.2.1 SCCS deltas

Early versions of the SCCS system stressed more heavily the possibility to combine
deltas freely. The intention seemed to be to use deltas in a way much like we are
using options. Apparently, this did not work very well, and some of the mechanisms
for evolving the delta structure were actually removed in later versions [Gla78].

If freely combining deltas did not work well in SCCS, why do we assume that freely
combining options in our COV tool will work better? The most crucial difference,
in our opinion, is that SCCS did assume too strongly that automatic merges are
possible. For instance, if a delta is inserted in the middle a version graph, the delta
must be compatible with all its successors. In our model, the equivalent action
would create a new set of versions, mirroring but distinct from the old “successors”.
Even though external mechanisms (like validities) may be used to express that those
versions are now the preferred ones, they may still be independently validated.

Importantly, merging unrelated deltas in SCCS does not provide any way of record-
ing changes which are local to the particular combination of those deltas (beside
creating a new, unrelated delta to hold them). In COV, the ambitions permit edit-
ing changes to be recorded for any combination of options. Conceptually, deltas
identify physical changes, while options identify logical changes—and COV per-
mits the latter to be implemented differently according to context.

8.2.2 P-EDIT and MVPE

The multi-version editors P-EDIT and MVPE [SBK88] exhibit a model (and imple-
mentation) which is very close to ours. They use a number of dimensions to classify
versions consisting of fragments. A number of versions identified by the edit set are
edited in parallel, and one of them, called the view is shown to the editor user at
one time. *Dimensions* correspond to options, the *edit set* to the *ambition* and the *view* to the *choice* in our model.

In this approach, dimensions seems to be used to classify a number of pre-created versions, as opposed to our options which describe the choice space of potential versions. Fragments belonging to the same version are linked in a so-called thread, while no such linking is performed in our system. There are no concept of validities or stability, and no mechanism for automatic validation of versions, such as in [Ha88]. Still, this work is close enough to ours that we ought to point out: Even though there are no early publications on COV, the ideas date back several years and thus are independent on Sarnak’s and Kruskal’s work.

### 8.2.3 Miscellaneous

It has come to our attention that there do exist one or two other configuration management/versioning systems which are focusing strongly on changes, and where versions are considered to be composition of changes. (One of them is even labeled “change oriented”.) What we have heard about them leads us to believe that “changes” are treated essentially the same as SCCS deltas, and thus will have the same weaknesses as discussed previously. Unfortunately, we do not have any publicised material to cite.

### 8.3 Continued Work

There is a fair number of “loose threads” to follow up on in this work:

- Controlling the complexity of the boolean expressions involved is one of the most crucial issues for the success of COV. In particular, more advanced functionality, like multi-version editing and analysis, puts heavy demand on our capabilities here. Experimenting with better heuristics for expression simplification may bring some leverage to those problems. Furthermore, it is still too early to rule out completely the possibility that there is some “smart” representation for the expressions which brings out patterns and possibilities for optimization that are hidden when all problems just are stuffed through the same general “meat grinder” expression simplifier.

- The connections to the software process remains to be completely traced out. The EPOS project contains a process management component, where a prototype is forthcoming, and where the integration with configuration management through COV is a main emphasis.

- Binary options is a fairly low-level mechanism for selection of versions. The concept of a high-level *configuration description* has been introduced, along with some of the ways it is used to interface to other components in the system (notifiers). However, a comprehensive design remains to be done.
• We have silently assumed that all options are global and potentially may affect all components in the database, and only briefly hinted at the possibility of limiting the scope of options. Such a mechanism appears to be convenient and important for practical use, and needs to be designed.

• Tools for editing and analysis in a multi-version context need to be designed and implemented. Without such tools, only parts of the potential of COV will be realized.

• Most important is certainly a trial implementation of basic change-oriented versioning, without too many bells and whistles, but robust enough to be used for real development over some period. After that, we should have a better grip on the complexity issues, and a better feeling for how far change-oriented versioning stands up to the promises for everyday use.
Appendix A

Prolog Interface to EPOSDB Prototype

A.1 General Principles

Most of the Prolog predicates are type-specific and generated from the database schema. The Prolog view of the database is very close to the default implementation of the Entity-Relationship data model over a relational database: For each entity type, there exists a property relation in the database with an object identifier as key, and one column for each attribute. In the case of subtyping, the property relations for a type and its supertypes are joined over the object identifier. Relationships are also implemented by relations, and the relation table in this case contains one object identifier column for each role in the relationship.

A.2 Database Query

For each entity type $E$, there exists a property relation $E(O, A_1, \ldots, A_n)$ in the database and the corresponding Prolog predicate $\text{qry}_E(O, A_1, \ldots, A_n)$. The predicate has a parameter $A_i$ for each attribute of the entity type $E$, but not for supertypes of $E$. (To query for supertype attributes, use the supertype predicate instead.) The first parameter $O$ is the object identifier for the object. All combinations of bound and unbound parameters is permitted.

- $O$ is bound: Retrieve attributes for object $O$, or test if object $O$ has attributes with specific values (one or more bound $A_i$’s). Succeeds once.

- $O$ is unbound: Retrieve objects with attributes matching bound $A_i$’s. Retrieves successive matching objects if backtracked over.

For relation $R$, the property relation is $R(O_1, O_2, A_1, \ldots, A_n)$, and the Prolog predicate is $\text{qry}_R(O_1, O_2, A_1, \ldots, A_n)$. Here the parameter list $A_i$ includes attributes of $R$ itself and all its supertypes, with supertype attributes first. $O_1$ and $O_2$ are
the object identifiers for the objects related by a given relationship. Usage is as for
entities. With $O_1$ bound and $O_2$ unbound, all objects related to $O_1$ with matching
relationships are successively returned in $O_2$, and vice versa. With both bound, the
predicate tests whether objects $O_1$ and $O_2$ are related by a matching relationships.

Note that the term $\texttt{qry}\_R(o, \ldots, \ldots)$ (bound $o$, anonymous variables for all attri-
butes) can be used to check whether the object $o$ is of (sub)type $R$. There might
also be a separate predicate to test the type of an object. (I.e. one which is capable of returning
the actual type of an object, as opposed to testing whether it is of a
particular type.)

The C backend is free to load and cache objects in virtual memory as they are
examined, in order to optimize later queries.

A.3 Database Update

For each entity type $E$, there is a Prolog predicate $\texttt{upd}\_E(O, A_1, \ldots, A_n)$, parameters
as for $\texttt{qry}\_E$. This predicate can be called with $O$ bound or unbound. If bound, the
tuple in the database relation identified by the object identifier $O$ is updated in all
attributes corresponding to bound $A_i$’s, while all $A_i$’s which are variables become
bound to the old and unchanged values of these attributes. If $E$ corresponds to a
type the object identified by $O$ does not currently have, the type of the object is
changed. For instance, if the object is of type $T$ and is updated with $\texttt{upd}\_S$ for a
subtype $S$ of $T$, the object is specialized to $S$. In this case, unbound $A_i$’s cause default
values to be inserted. Default values are determined according to the domain of the
attribute, e.g. 0 for integers, empty strings for strings etc.

If $O$ is unbound, a new object is created and $O$ becomes instantiated to its object
identifier. Tuples in the property tables corresponding to supertypes of $E$ are created
and initialized with default attribute values.

The predicate fails if backtracked and also if the requested update would lead to an
inconsistent database state.

The predicate $\texttt{cvt}\_T(O)$ converts the object identified by $O$ to the type $T$. This
predicate must be used to generalize an object to one of its supertypes (i.e. remove
subtype part).

All updates are performed (with regard to update of visibilities) as if the object in
its entirety was loaded into virtual memory, updated and written back. (See figure
7.7 on page 131 for the algorithm.)

Relationships do not have object identities and must be handled differently. For
relationship type $R$, there are two update predicates:

$\texttt{ins}\_R(O_1, O_2, A_1, \ldots)$
$\texttt{del}\_R(O_1, O_2, A_1, \ldots)$

for inserting and deleting matching relationship instances, respectively. Both predic-
ticates must be called with $O_1$ and $O_2$ instantiated. There may be several relation-
ships of the same type connecting a given pair of objects, so the two object identifiers \( O_1, O_2 \) does not identify a relationship uniquely. Uninstantiated \( A_i \)'s in \( \text{ins} \cdot \text{R} \) are filled in with default values. \( \text{del} \cdot \text{R} \) deletes all matching relationships.

### A.4 Working Context and Transaction Management

The predicate \( \text{start} \cdot \text{transact}(T, A) \) starts a new long transaction by selecting an ambition for it, and will (in the first version of the database) fail if there is another transaction with overlapping ambition. \( A \) is a list of pairs \( [\text{option} | \text{value}] \) where \( \text{option} \) is an option name (Prolog atom) and \( \text{value} \) is either of the Prolog atoms \( t \) or \( f \). \( T \) is a transaction identifier (an atom) which is returned from the database system, and subsequently can be used to identify the transaction.

\( \text{set} \cdot \text{choice}(C) \) sets the (version) choice. \( C \) is a list of the same format as \( A \). Only options not specified in the ambition can be set here, thus constraining the choice to be contained in the ambition. Options not specified in either \( A \) or \( C \) are set to their default values.

\( \text{set} \cdot \text{choice} \) may be called several times during the same transaction. In that case, the previous choice forms the default for options not mentioned in the new choice. The first time \( \text{set} \cdot \text{choice} \) is called in a transaction, the default values are taken from the option table in the database.

An application (operating system process) may disconnect from a long transaction by calling \( \text{disconnect} \cdot \text{transact} \). The choice setting persists even when a transaction is disconnected. Use \( \text{connect} \cdot \text{transact}(T) \) to connect to an existing transaction. If the application was already connected to a transaction, \( \text{connect} \cdot \text{transact} \) implicitly disconnects it before connecting to a new one. Only one application can be connected to a given transaction at the same time. The current transaction is terminated by calling \( \text{end} \cdot \text{transact} \) or \( \text{abort} \cdot \text{transact} \). (In the first version of the database, \( \text{abort} \cdot \text{transact} \) is identical to \( \text{end} \cdot \text{transact} \), since there is no way to discard the changes done during a long transaction.)

To query and manipulate options, there are two predicates similar to the ones for querying and manipulating objects. (In fact, options can be said to be entities in their own right, although not versioned.)

\( \text{qry} \cdot \text{option}(O, N, D) \) can be used to query the attributes of a particular option \( O \). At present, there are only two, name and default value. \( O \) is a unique, system-generated identifier for an option, and may be taken from the same space as object identifiers.

\( \text{upd} \cdot \text{option}(O, N, D) \) can be used to change attributes or create new options. For both predicates, \( N \) and \( D \) are may be unbound or bound to Prolog atoms, and the semantics are as for the corresponding predicates for entities.
A.5 File Manipulation

In the database, primary and derived files are represented as Unix files and directories in a "file pool". Primary files are represented as files using the standard change-oriented embedded delta format, while derived files are represented as directories. These directories contain a file for each stored version of the derived file. The name of each file is the predicate number (in ASCII) of the visibility for that particular version. The timestamp of the file is likewise the timestamp for that version. (This is perhaps redundant, as timestamps will be stored as object attributes also.)

The pathnames used, both within the file pool and the workspaces, are under the control of the Prolog application. Some reasonable scheme should be used to avoid huge directories in the pool (which are slow to search), e.g. two or three levels of directories.

\textbf{checkout\_file}(P, W), where P and W are Prolog atoms, respectively a pathname in (i.e. relative to the root directory of) the file pool and a pathname in the workspace:
Check out a primary or derived file from the pool into a workspace file. The predicate fails silently if the file corresponding to P does not exist in the file pool, or if it is derived, it is not defined for the current choice.

\textbf{checkin\_primary}(P, W): Check in a primary file to the file pool. Create the file P if it did not exist.

\textbf{checkin\_derived}(P, W, V): Check in a derived file to the file pool. V is the visibility to use for this file version. If V is unbound, the choice is used as the visibility. See the next section for representation of predicates.

Both \textbf{checkin} predicates fail if the workspace file does not exist.

A.6 Boolean Expressions

An application need not normally consider visibilities and other boolean expressions. An exception is if it is going to use the V parameter of \textbf{checkin\_derived} to specify a larger visibility. In that case, the visibility should be computed as the intersection (logical AND) of the visibilities of all objects the derived object depends on.

Boolean expressions are abstract datatypes, which to the application are represented by the indices they have in the predicate table internal to the database. The constant expressions \textbf{True} and \textbf{False} are represented by the constants (Prolog atoms) \textit{t} and \textit{f}.

New expressions can be computed by the predicate \textbf{compute\_exp}(E, I), where \( E \) is bound to a Prolog term representing the expression and \( I \) is unbound and returns the predicate index. The representations accepted for \( E \) are:

\begin{itemize}
  \item \textbf{and}(E_1, E_2)
  \item \textbf{or}(E_1, E_2)
\end{itemize}

142
- \texttt{not}(E_1)
- Expression index
- Option/value lists (as for ambitions)

\texttt{obj\_visibility}(O, V) called with an object identifier \(O\) returns in \(V\) the (index for the) visibility of the current version of the object, i.e. the set of choices for which the version of the object is the same. This is computed as the intersection of the visibilities for the individual fragments of the object which are present in the current version, i.e. those visibilities which are true.

\texttt{file\_visibility}(F, V) computes the visibility of a file version, in the same way as \texttt{obj\_visibility}. \(F\) is the name of a primary or derived file in the file pool.

Since relationship instances cannot be uniquely identified by an object identity, an application interested in visibilities for relationships must use a variant of the query predicates: For each relationship \(R(E_1, E_2)\) with \(N\) attributes, there is both a \(n+2\)-ary predicate \texttt{qry}\_\(R\)(\(O_1, O_2, A_1, \ldots, A_n\)) and a \(n+3\)-ary predicate \texttt{qry}\_\(R\)(\(O_1, O_2, V, A_1, \ldots, A_n\)). These predicates behave as the corresponding predicates for entities, except that for the last alternative, the visibility for the corresponding database tuple is returned.

The expression indices returned by predicates in this section are unique and persistent only for as long as an application remains connected to the same long transaction. I.e. they should not be stored across calls to any of the \*\_\texttt{transact} predicates.
Appendix B

Schema for Database Prototype

B.1 Data Definition Language

The data definition language is defined by the following (simplified) Yacc grammar.

```yacc
%token IDENTIFIER ZERO_DOTDOT ONE_DOTDOT
%start defs

%%

defs :
    | def defs

    

def :
    entity_def
    | relationship_def

    

entity_def :
    'Entity' entity_name supertype_clause attribute_list ';'

    

entity_name : IDENTIFIER ;

relationship_def
```
: 'Relationship' relationship_name supertype_clause
  relates attribute_list ';'
;
relationship_name : IDENTIFIER ;

supertype Clause
  :
    | 'Is' entity_or_relationship_name
;

type_or_relationship_name : IDENTIFIER ;

attribute_list
  :
    | '{' attributes '}'
;

domains
  :
    | attribute ';' attributes
;

attribute
  : attribute_name ':' attribute_domain
;

attribute_name : IDENTIFIER ;

attribute_domain : IDENTIFIER ;

relates
  : '(' role ',', role ')'
role
  : entity_name cardinality
  ;

cardinality
  :
    | "," lowbound highbound
  ;

lowbound : | ZERO_DOTDOT | ONE_DOTDOT ;

highbound : '1' | 'N' | 'M' ;

A schema definition consists of a list of entity and relationship definitions. Each
definition starts with the keyword Entity or Relationship, followed by the name
of type defined. If the type is a subtype, the keyword 'Is' introduces a list of
supertypes which it inherits from.

For relationships, the roles are enclosed in parentheses. Each role names an entity
type and specifies a cardinality range. Cardinalities may be specified as min..max,
as min.. (maximum cardinality unspecified and defaults to N, i.e. unbounded), or as
max (no ellipsis, minimum cardinality defaults to zero). For each role, the cardinality
expresses how many different objects may occur in this role for one object in the
other.

Note that existence dependencies may be expressed by having non-zero minimum
cardinality.

Both relationship and entity definitions take an (optional) attribute list. The possible
attribute domains are not specified in the grammar. In the schema listing below,
the domains are specified informally.

B.2 Schema Listing

B.2.1 Entities

Entity part {
    dyn_attributes : list of [ name | value ] pairs;
};

Entity subsystem Is part ;
Entity component Is part {
  primary: prolog atom 't' or 'f';
  contents: prolog atom (filename relative to file pool);
  versioned: prolog atom 't' or 'f';
  timestamp: integer (Unix timestamp, seconds);
};

Entity typedesc {
  description: prolog string ;
};

Entity tool {
  matcher: prolog atom ;
};

Entity derivation Is subsystem {
  analyzed: integer (Unix timestamp) ;
  cmd_line: string ;
};

B.2.2 Relationships

Relationship composition (subsystem : N , part : M) {
  name: prolog string ;
};

Relationship exported Is composition ;

Relationship is_a (typedesc : 1..1 , part : N) ;

Relationship executable (tool : N , component : 0..1) ;

Relationship toolparam (tool : N , typedesc : M) {
  paramname, pattern: prolog string ;
};

Relationship toolin Is toolparam ;

Relationship toolout Is toolparam (tool : 1..N, typedesc : M) ;

Relationship imports (subsystem : N, part : M) ;
Relationship derivin Is
  composition (derivation : N, part : M) ;

Relationship derivout (derivation : N, part : M) ;

Relationship derivvia Is
  derivout ( derivation : 0..1, part : M) ;

Relationship deriver (derivation : N, tool : 1..1) ;

B.3 Semantics of Schema

B.3.1 Composition structure

A subsystem is a name space where parts may be entered and named, relative to the subsystem. Parts have no absolute name beside their object identity.

Note the similarity to a hierarchical file system. Each subsystem corresponds to a directory, while a part corresponds to the union { file, directory }. Names are associated with the composition relation. This gives a semantic of names very similar to the Unix file system, where a file may have different names in the different directories where there is a link to it. A further parallel with the two-level Unix file system is seen in that the object identity corresponds to the inode number.

This similarity is intended to facilitate modelling of traditional tools like e.g. compilers designed to work in a traditional hierarchical file system. "Include"-directories, libraries, module interfaces etc. may all abstractly be seen as name spaces, and the name space used to bind a name is defined through tool-specific or even dynamic rules. In traditional tools, such name spaces are conventionally modelled as directories.

The contents field contains the name of a file which is used to actually store the contents of a component. Our prototype must be open-ended enough to be able to handle system libraries, system tools etc. external to the database without actually storing these in the versioned database. We intend to represent them as components, in effect considering them as atomic from our point of view. The attribute versioned of a component specifies whether the filename (contents) refers to a versioned file internal to the database or to such an external file/directory. If components such as operating system or compiler runtime libraries are to be versioned, they have to be stored inside our version archive.

For versioned components, contents is a system-generated file name relative to a "file pool" internal to the system and not intended to be accessed directly by tools. Versioned components come in two flavors, primary and derived (binary). The contents of primary objects are stored in the file using the delta mechanism described in section 5.6.2 and with the same conventions for checkin. The contents
of derived objects is a file cache, where versions are installed with a specified visibility and may be purged.

There is a subtype exported of the composition relationship type. Names associated with an exported link are exported by the subsystem S, i.e. can be referred to from other subsystems importing S.

While components have an explicit timestamp (the last modification time for the contents), subsystems have a timestamp implicitly defined as the maximum of the timestamp of all its components. (This is one case where a full object-oriented model with methods would be convenient.)

It is possible to associate arbitrary descriptive attributes with parts through the dyn.attributes attribute. Note the overloading of the word "attribute" here. The type descriptors could contain information about mandatory or permitted attributes for parts of specific types.

### B.3.2 Component and tool types

All parts have an associated typedesc. Logically, typedescs describe subtypes of component and subsystem. However, the set of component and subsystem types is large and dynamic. Our database system does not allow dynamic schemas, and for our purpose, this flexibility is not really needed anyway, since a single database type suffices to represent all components and subsystems.

The functional properties of tools are represented by the tool entity. The actual executable program implementing these functional properties is a part which is related to the tool object through the executable relation. Several tools can share the same executable, i.e. the same executable can have different functions depending on which command line it is invoked with. (This does to a certain degree, but not fully satisfactory, model the Unix convention of having different names, i.e. links, for the same executable program, performing different functions.)

A useful analogy is to compare the tool entity with an interface, and the part it is related to with an implementation of that interface.

The toolparam relation and its subtypes are used to describe the parameter bindings for each tool. Each parameter has both a name and a pattern used when tools and objects are matched in order to plan automatic rederivations, for instance for a C-compiler:

\[\text{toolin(paramname = "source", pattern = "$\{file\}.c")}\]
\[\text{toolout(paramname = "object", pattern = "$\{file\}.o")}\]

(The modelling of the component type/name suffix relationship might be made more explicit.)

Candidate matchings determined by the generic matcher above are fed to the tool-specific matcher (matcher). This procedure is the one that actually fills in the derivation step node, determines all dependencies and builds the command line to
invoke the tool—or fails, if it determines that the object cannot be derived this way after all. In principle, all matching could be done by matcher, but the “generic” matching on the types and names of the parameters is used as a coarse filter to screen out obviously non-applicable candidate tools.

### B.3.3 Derivations

![Diagram of derivations example]

Figure B.1: Derivations example

The derivation entity is used to model derivation steps, and can be seen as an instantiation of a tool, which it is related to through the deriver relation. A derivation is a subtype of subsystem, i.e. an aggregation of its input objects. In particular a derivation has an implicitly defined timestamp, which is the maximum of the timestamps for the input objects, just as other subsystem nodes. Derivation
nodes are created during backwards chaining in the activity planner (but could also be set up manually or by other tools a priori).

There may be more than one way to derive a given object, derivout is N:M. The derivation path that was actually taken the last time the object was derived is indicated by specializing one of the derivout arc to derivvia. (Note how subtyping is used to restrict the derivvia relation!) Only this arc is involved in determining whether the object is up-to-date.

We have defined the visibility for a version of an object to be the intersection of all true visibilities of fragments of this object. (This definition corresponds to the definition of visibility for derived objects stored in the derived object cache.) tool.matcher could in principle be arbitrarily “smart” and set visibilities on the dependency arcs based on a fine-grained analysis of visibilities in the source objects. It has to ensure that the visibilities on the input arcs are “large enough” compared to the visibility of the derivation objects they leave: If a derivation object is visible in a given version, the other tools have to be able to rely on all relevant dependencies for this derivation step being visible. A simple way to ensure this:

- Compute the intersection of the visibilities of all object versions the dependency analyzer has to examine.
- Use this (or a simplified subset, see section 5.4.5) as the visibility of both the derivation object and the input arcs.

To decide whether a derived object version is up-to-date, one will first have to examine whether the dependency information is up-to-date, by comparing the timestamp of the derivation object (as a subsystem) with derivation.analyzed. If it is not up-to-date, the matcher has to be run again, and if the dependencies then change, the stored derived object version must be discarded and rederived. Otherwise, its timestamp can be directly compared with the timestamp of the derivation object.

In the example in figure B.1, the ordering of the timestamps is such that \( ti < tj \) iff \( i < j \). The relocatable object a.o is derived from a.c, and is up-to-date with respect to both a.c and a.h. It is out of date with respect to a.p, but this does not matter since it is currently not derived from this object. Also, the derivation node for the Pascal compilation has to be reanalyzed for dependencies.
Bibliography


Index

abstract datatype, 26
activity manager, 108
addition,
of option, 65
ADT, 26
aggregation, 16
ambition, 54, 60
complexity of, 82
interaction with stability, 64
primary, 78
ambitions,
overlapping, 114, 72, 82
analysis methods, 8
atomic objects and versioning, 34
attribute,
composite, 50
attribute constraint, 37
attribute domain, 18
attribute,
version, 37
attributes,
EPOSDB, 101
backward delta, 33, 43
behavioral object-orientation, 20
binary data model, 15
binary data models,
COV of, 95
boolean expressions,
simplification, 78
bound configuration, 51
build dependency, 108, 37, 44
C compiler,
modelling of, 117
caching,
derived objects, 38
change propagation,
within workspace, 107
change-oriented versioning, 10, 33, 51
applicability of, 90
checkin,
EPOSDB, 105
nested transaction, 84
checkin/checkout, 24, 60, 74
checkout,
complexity of, 81
EPOSDB, 105
choice, 53, 56, 61
inherently non-valid, 64
choice space, 71
class,
EPOSDB, 99
clustering, 16
COBOL, 30
CODASYL DML, 30
commit,
of edit task, 62
of long transaction, 62
complete choice, 56
complex object, 16, 36
complex objects,
EPOSDB, 101
component, 36
compont role, 101
components,
derived,
caching of, 111
composite attribute, 50
composite object, 16
concurrency object, 21
COM context, 71
optimistic, 122
timestamp-based, 75
config, 40
configurability, 39
configuration, 36
configuration descriptions, 66, 74
configuration management, 32, 36, 51
configuration selection, 37
configuration,
configuration (continued):
  bound, 51
consistency, 19
consistency constraint,
  in EPOSDB, 125
constraint, 19
  attribute, 37
contractual approach, 7
copyin, 114
copyout, 114
copyout/copyin,
  EPOSDB, 106
correctness, 59
COV,
  experiences with, 129
currency indicator, 29
cursor, 29
DAG, 33
Damokles, 49
data integration, 12
data manipulation language, 26
data model,
  definition, 13
  hierarchical, 14
  network, 14
  object oriented, 15
  relational, 14
  value oriented, 15
data models,
  binary, 15
  classical, 14
database integration, 9
database system, 12
database table, 120
datatype,
  abstract, 26
DBMS, 12
deadlock, 22
delta, 43
  embedded, 33
  forward/backward, 33, 43
  minimal, 90
  size, 130
delta storage, 33
deltas,
  embedded, 43
dependency (continued):
  build, 37
  fine-grained, 38
derivation, 37
derivation graph, 115
derivation,
  in EPOSDB, 115
derived components,
  caching of, 111
derived object, 116
derived objects,
  EPOSDB, 107
design methods, 8
design,
  object-oriented, 8
differencing, 76
text files, 90
distributed environments, 40
DML, 26
domain, 18
DSEE, 45
edit task, 60
  committing, 62
embedded delta, 33
embedded deltas, 43
embedded SQL, 29
encapsulation,
  of operations, 19
engineering DBMS,
  requirements, 13
entities,
  subtyping of, 123
entity,
  EPOSDB, 99
Entity-Relationship model,
  COV of, 96
environments,
  distributed, 40
EPOS data model,
  implementation, 122
EPOSDB, 110
EPOSDB data model, 99
EPOSDB,
  database implementation, 120
derivation, 115
  planning, 116
  product structure, 114
object (continued):
  composite, 16
  generic, 32
object identity, 15
object identity and versioning, 34
object,
  version of, 56
object-orientation, 20
  behavioral, 19
  full, 20
  structural, 20
object-oriented data models,
  COV of, 98
object-oriented programming, 8
operating system,
  configurability of, 40
operations,
  EPOSDB, 102
  generic, 19
  user-defined, 19
optimistic concurrency control, 122
option, 55
  local, 77
option validity, 58
option,
  addition of, 65
ordering of fragments, 57
Orion, 50
overlapping ambitions, 114
  synchronization of, 72
PACT, 48
PCTE, 48
persistent programming languages, 30

planning,
  in EPOSDB, 116
preference, 37
preprocessor, 29
primary ambition, 78
process,
  change-oriented versioning, 60
process modelling, 7
process programming, 7
product structure,
  in EPOSDB, 114
programming environments, 9
programming languages (continued):
  persistent, 30
programming,
  object-oriented, 8
  transformational, 8
property table, 123
prototyping, 7
RCS, 42
rebuild, 37
recompilation,
  smart, 38
reconciliation,
  conflicting updates, 75
relation, 100
  EPOSDB, 100
  non-first-normal-form, 16
relation tables, 124
relational algebra,
  versioned, 94
relational data model, 14
  versioning of, 34
relational database,
  COV of, 91
relationship, 100
  versioning of, 35
revision, 32
SBDM, 95
SCCS, 42
schema, 14
schema evolution, 20
scope,
  of options, 86
serializability, 22
simplification of expressions,
  complexity of, 129
size of delta, 130
smart recompilation, 38
  EPOSDB, 108
software,
  appetite for, 6
software configuration management,
  36
software crisis, 6
software engineering, 6
software engineering environments, 9
spiral model, 7
SQL, 29
  embedded, 29
SQL views, 121
stability, 83
  of versions, 58
StP, 113
structural object-orientation, 20
subdatabase, 72, 74
subobject, 101
subtype, 17
superobject, 101
synchronization,
  option-based, 77
table, 120
task stability, 83
text files,
  COV of, 90
text move, 57, 91
timestamp, 122, 37, 44
timestamp-based concurrency control, 75
tool description,
  in EPOSDB, 115
tool descriptions,
  in EPOSDB, 117
tool integration, 8
total validity, 58
traditional database systems, 12
transaction,
  in EPOSDB, 111, 114
  long, 23, 72
transaction management,
  implementation of, 120
transaction mode, 76
transactions,
  COM context, 71
  long, 22
  nested, 25
transformational programming, 8
trigger,
  EPOSDB, 105
triggers, 19
type refining, 105

update conflicts,
  resolution, 24
user-defined operations, 19
validity, 53, 58
  complexity of, 85
  inherent, 59
variability, 39
variant, 32
version graph, 32
version group, 32
  criteria for forming, 34
version merging, 114
version of object, 56
versioning, 21, 32
  change-oriented, 10, 33
  EPOSDB, 102
  implementation of, 121
  transparency of, 35
version-oriented model,
  emulation of, 87
version-oriented versioning, 51
  applicability, 34
visibilities,
  complexity of, 129
  update of, 127
visibility, 57
  complexity of, 79
  intersection of, 84
  update of, 62

waterfall model, 7
workfile, 42
workspace, 110, 114
   EPOSDB, 106