CAMERA
Support for Distributed Cooperative Work

CAMERA: Ondersteuning voor Gedistribueerde Samenwerking
(met een samenvatting in het nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Rijksuniversiteit te Utrecht
op gezag van de Rector Magnificus, Prof. Dr. J.A. van Ginkel
ingevolge het besluit van het College van Dekanen
in het openbaar te verdedigen
op maandag 19 oktober 1992 te 12.45 uur

door

ERNST LIPPE

geboren op 25 April 1959,
te Den Haag
promotor: Prof. Dr. S.D. Swierstra,
Faculteit Wiskunde en Informatica

This research was supported by the Software Engineering Research Centre.
CAMERA

Support for

Distributed Cooperative Work
Contents

Preface ix

1 Introduction 1
  1.1 Requirements ........................................... 2
    1.1.1 Repository requirements ......................... 3
    1.1.2 Version management requirements ................. 4
    1.1.3 Consistency management requirements ............. 8
    1.1.4 Cooperation requirements ......................... 10
    1.1.5 Distribution requirements ......................... 12
    1.1.6 Miscellaneous requirements ....................... 12
  1.2 Existing systems and practises ..................... 13
    1.2.1 Version management practises ..................... 14
    1.2.2 SCCS and RCS .................................. 16
    1.2.3 Derivation managers ............................. 17
    1.2.4 Loosely-coupled development of programs .......... 18
    1.2.5 NSE ............................................. 19
    1.2.6 DSEE ............................................ 20
    1.2.7 Adole ............................................ 20
    1.2.8 Using file-systems ............................... 21
    1.2.9 Object Management Systems ....................... 22
    1.2.10 Historical data-bases ............................ 23
    1.2.11 PCTE ........................................... 23
    1.2.12 EPOS .......................................... 24
    1.2.13 KBSA .......................................... 24
    1.2.14 The Workshop System ............................. 24
  1.3 Conclusions ........................................... 25
    1.3.1 Expressive data storage ........................... 25
2 The anatomy of CAMERA

2.1 The OMS ............................................ 35
2.2 Version control in OMSes .......................... 36
   2.2.1 Version control on multiple objects .......... 37
   2.2.2 Problems with distributed version-controlled OMSes . 38
2.3 Snapshots ............................................. 38
2.4 The Snapshot OMS and environments ............. 41
2.5 The Album OMS ........................................ 44
2.6 Communication and distribution .................. 45
2.7 Working with CAMERA ................................. 47
   2.7.1 The OMS ......................................... 47
   2.7.2 Using environments ........................... 48
   2.7.3 Development policies .......................... 49

3 The CAMERA OMS .................................. 51

3.1 Introduction ......................................... 51
   3.1.1 The data model of the OMS ................. 52
   3.1.2 Class definitions .............................. 55
   3.1.3 Types ............................................. 56
   3.1.4 Relations ......................................... 57
   3.1.5 Methods .......................................... 58
   3.1.6 Method invocation .............................. 59
   3.1.7 Triggers ........................................... 61
3.2 The language interface ............................... 62
3.3 Built-in behaviour ................................... 63
   3.3.1 Creating objects ............................... 63
   3.3.2 Creating classes ............................ 64
   3.3.3 Relation-operations ......................... 64
   3.3.4 Naming objects ................................ 65
3.4 The Album OMS .................................... 66
3.5 Conclusions .......................................... 67
4 Data-structures for integral version management 69
  4.1 Introduction ........................................... 69
  4.2 A model for integral version management ............... 69
  4.3 The implementation model ................................ 71
  4.4 Linear version history .................................. 74
      4.4.1 Sparse index ........................................ 74
      4.4.2 Versions of sets .................................... 75
  4.5 Non-linear version history ................................ 76
      4.5.1 Storing versions of objects ......................... 76
      4.5.2 Implementations for versioned sets ............... 80
  4.6 Conclusions ............................................. 87

5 Derivation management 89
  5.1 Introduction ............................................ 89
  5.2 Derivation management ................................... 89
      5.2.1 Requirements ....................................... 91
  5.3 Existing systems ........................................ 93
  5.4 Derivation management in the CAMERA OMS ............. 97
      5.4.1 Representing derivations ......................... 98
      5.4.2 Example ........................................... 99
  5.5 Dependency maintenance .................................. 100
      5.5.1 The dependency set ................................ 100
      5.5.2 Example ........................................... 101
      5.5.3 Applications of dependency management .......... 103
  5.6 Equivalence of derived values .......................... 104
  5.7 Cache management ....................................... 105
  5.8 Summary and open issues ................................ 106
      5.8.1 Open issues ....................................... 107

6 Pieces 109
  6.1 Introduction ............................................ 109
  6.2 Requirements .......................................... 109
  6.3 Existing approaches ..................................... 111
  6.4 Pieces in CAMERA ....................................... 112
      6.4.1 OMS definitions ................................... 115
  6.5 The transplant operation ................................. 116
# Contents

6.6 Example .................................................. 117
6.7 The piece editor ........................................... 118
   6.7.1 Using the piece editor .............................. 120
   6.7.2 Modifying a piece .................................. 121
   6.7.3 Organisation of standard components ........... 121
6.8 The rules .................................................. 122
   6.8.1 The rule mechanism ............................... 123
   6.8.2 Examples .......................................... 125
6.9 Discussion ............................................... 127
6.10 Conclusions ............................................. 128

7 Merging ..................................................... 129
   7.1 Introduction .......................................... 129
   7.2 Existing merge tools ................................ 129
      7.2.1 Text-based merge tools ......................... 129
      7.2.2 Other merge tools ............................. 130
      7.2.3 Deficiencies ................................... 131
   7.3 Design goals ......................................... 132
   7.4 Operation-based merging ............................. 132
      7.4.1 Advantages of using operation-based merging 134
   7.5 Overview of the merge process ...................... 134
   7.6 Step 1: Forming raw blocks ........................ 135
   7.7 Step 2: Finding and solving conflicts .............. 140
   7.8 Computation of commutation behaviour ............ 141
      7.8.1 Computation of global commutation information 142
      7.8.2 Computation of local commutation information 142
   7.9 Problems with operation-based merging ............ 142
   7.10 Conclusions .......................................... 144

8 The CAMERA prototype .................................... 145
   8.1 Overview ............................................. 145
   8.2 Process architecture ................................ 145
   8.3 Unix compatibility .................................. 147
      8.3.1 A file-system view of the OMS ................. 148
      8.3.2 Implementation of the file-system view ....... 148
      8.3.3 Side-effect-free behaviour                  149
Preface

The work that is described in this thesis was performed at the Software Engineering Research Centre. First of all, I would like to thank Gert Florijn for inspiring discussions and support. The work presented here has of course also been influenced by the other members of the CAMERA team: Doaitse Swierstra, Atze Dijkstra, who was responsible for the implementation of the prototype, and Norbert van Oosterom. Over the years, several students have participated in the project: Jaco de Vroed, Arjan van Dijk, Leo Soepenberg, Klaas Eikelboom, and Kees van Schaik.

SERC provided a stimulating environment for doing research. I would like to thank all members of the SERC crew.

During their stay at SERC I have had interesting discussions with Geoff Clemm, John Nicol and Jan Wielemaker.

Several people helped in reading (parts of) my thesis: Liesbeth Dusink, Erik Nieuwland and Gert Florijn.

Finally, of course thanks to Liesbeth and Coby for care, maintenance and moral support.
1 Introduction

Computer supported cooperative work is becoming increasingly important. Good support systems for this type of work are slowly becoming available. One aspect is the support for managing changes in the data that is modified as a result of the activities of the participants. These modifications will lead to multiple instances of some data items, called versions. Versions can arise due to parallel activities, when several users each create a new instance, or through historical development, when a data item is modified a new version of it is created. Version management is the discipline that attempts to deal with versions in a systematic way. Version management is important for cooperative work because it manages the evolution of the data that is the subject of these activities.

Cooperative work is sometimes performed on one central computer or on a tightly-coupled local area network of computers. However, there are also many cases where cooperative work must be performed on a loosely coupled distributed network. Loosely-coupled distributed networks are networks where connections between network sites may be slow or temporarily absent. There are many examples of loosely-coupled distributed environments. A first example is that of users on different sites that can connect with a modem over telephone lines. Permanent telephone connections are frequently not available due to high prices. Because telephone rates are lower during off-peak hours, systems may prefer to use the night hours for exchanging data. The transmission rate of telephone lines is not very high. This makes it difficult to exchange large amounts of data. Another example of loosely-coupled distributed environments, is a collection of personal computers without network. Communication between computers is normally done by transmitting floppy disks. Loosely-coupled networks can also arise when personal computers are used together with a network of tightly connected computers. Users may want to take some of their work home to continue their work on their own computer. The same occurs with lap-top computers that are used while travelling.

A DCWSS
Cooperative work in loosely-coupled environment requires special support. In this chapter we use the term Distributed Cooperative Work Support System (abbreviated DCWSS) for a system that supports cooperation in loosely-coupled distributed environments. The subject of this thesis is the design of a DCWSS (called CAMERA) that can be used in such environments.
To give the reader some intuition about the kind of system we had in mind, we start with a very brief outline. The following section (1.1) goes into more detail and contains a list of requirements for the system.

We wanted to design a general purpose system that supports many different kinds of development activities, e.g. program development or collaborative writing. The system itself does not provide the applications, such as compilers or text formatters. It only provides building blocks that can be used by these applications.

One of these building blocks is a repository for storing the user’s data. Different applications have different requirements for such a repository. The repository should be sufficiently powerful to accommodate a wide range of applications. It must also be extensible so that new applications can be added.

During development new versions of data items are created. Version management facilities record the history of data, and can be used to trace the development of individual data items.

Cooperative work requires support for the exchange of data among the participants. Facilities should be available to make these exchanges run smoothly.

One aspect of these exchanges is the physical transport of data between different sites. In a loosely-coupled distributed network connections between different sites can be slow or absent. This complicates data transport. On one hand communication cannot always be synchronous, communication requests may be queued for later execution. The possibly low bandwidth of communication lines adds further difficulties. Compression of the transported data may be necessary to achieve adequate performance. Different transport media can be used as the underlying transport mechanism: standard network protocols like TCP/IP, modem connection over telephone lines, electronic mail, and floppy disks. All these physical transport mechanisms have different characteristics, but the system should hide these whenever possible.

Another aspect of exchanges is the logical exchange. There are usually certain (implicit) rules among developers about these exchanges. These rules will describe which data items will be exchanged, when they are exchanged, and how the data will be integrated by the receiver. The DCWSS needs mechanisms that can be used to automate these operations.

This chapter examines the problems with development in loosely-coupled environments, and provides a short survey of existing solutions. First, in section 1.1 a list of requirements for a DCWSS for loosely-coupled environments is given together with a short rationale. Then section 1.2 describes some existing systems, and section 1.3 examines in what respect they can meet these requirements.

Later chapters contain a description of the CAMERA system.

1.1 Requirements

Working in a loosely-coupled distributed environment causes specific problems. In the following sections we examine some of these problems and distill a set of requirements for
Requirements

a version management system that supports development work in such an environment. Some of these requirements are of course also desirable in any development environment. In a loosely-coupled system they become even more important because communication between people in such an environment gets more difficult.

The requirements are illustrated with an example development in which two persons, Jack and Jill, are developing and maintaining a small software system. This software system contains a few programs and some documentation. The documentation is written in \TeX [Knuth84] and includes pictures that have been made by a separate drawing program.

Each developer works independently on some components of the system. Regularly they exchange some of their developments.

Terminology

We use the term components to describe individual data items. In this discussion components are similar to files. A set of components that forms a conceptual unit and is the subject of the development activities is called a product. A module is a component that contains the source for a part of a program source. Every user has a workspace which contains a collection of components. A tool is a program that is used to manipulate components.

1.1.1 Repository requirements

1 Expressive data storage

The data storage system should have sufficient expressive capabilities. It must be able to describe and manipulate properties of the components and their relationships.

Components have certain attributes that describe them. One property of a component is when it was created and by whom. Other attributes describe the status of the component, e.g., has this component already been tested. When these properties are represented explicitly in the storage system, tools can use this information.

Components are related to one another. A component that contains a compiled program is related to its sources. The program is also related to its documentation. By representing this information explicitly in the storage system, it helps the user in organising and manipulating components. Browsers use it for examining the sources and documentation for the programs in the system.

Different types of components can be distinguished. Some components contain program sources in a specific programming language. Other components contain compiled programs, documentation, etc. Different types of components have different kinds of operations associated with them. For example, the “edit” operation on a program text could invoke a syntax directed editor. For a picture component this operation would invoke the drawing program. Components of the same type will have the same internal structure and attributes. When the type of a component is represented explicitly in the repository, the system can use this information for enhanced support. Representing typing information
in the storage system makes it possible to do type-checking, e.g. the system could warn when the Pascal compiler is invoked on a \TeX-source.

1.1.2 Version management requirements

2 Identifiability of components

Each component must have a name that can be used to identify it. In a centralised (i.e. non-distributed) environment this can be easily achieved, since there frequently is only one component with a given name. In a centralised environment name clashes can be detected easily. In a loosely-coupled distributed environment, this is much harder because it is frequently impossible to determine whether a given name is already used at a different site. In our example Jack could create a new component called "IO.c" that contains routines for doing low-level file input output, while Jill creates a component "IO.c" with routines for communicating with the user via a fancy window based interface. In this case there is a name clash.

The names of common components must be shared among the users. If Jack renames a certain "IO.c" to "interface.c", while Jill continues to use the old name to refer to it, they have a problem when exchanging and using each others components. Therefore, such name changes must be propagated to all users of this component.

3 Identifiability of versions

Different versions must be clearly identifiable; each version must have at least one name that can be used to uniquely identify this particular version.

Components are subject to change, thus at any given point in time there can be many versions of one component. These versions have to be distinguishable, otherwise all kinds of disasters can happen. For example, during communications the newest version could be accidentally overwritten with an older one. A user might accidentally delete the most recent version in the assumption that it was an old one. It is also possible that a user accidentally starts editing an old version which leads to duplication of work.

It must also be easily determinable whether two components are versions of the same component. For example, if we have "IOa.c" and "IOb.c" it is not clear whether these are versions of a common component (e.g. "IO.c") or not.

4 Trace-ability of versions and components

Information about the properties of a version and its derivation history must be recorded. Such information is needed to select a specific version and to understand why and how this version reached its current state.

Information about the development history of components is needed. New versions are almost always created by modifying existing versions. The new version is called a successor of the original version. The successor relation records the ancestry of components. Apart from the successor relation it is also important to record for a component who
created it and for what purpose. Development history information helps understanding the present state of this component and also points to the persons responsible for the changes. Knowing the date of the last modification also helps determining whether this version is the most recent one.

This version information is intrinsically tied to the component itself, so when a component is transmitted between different sites the version information for this component must be transmitted too.

5 Immutability

Historical versions must be immutable.

In loosely-coupled environment it is not possible to have only one copy of a specific version, because this version may be inaccessible from other sites. Therefore, there will be a copy of this version on all sites that need it. Another reason for the existence of multiple copies is the fact that users frequently copy a version to their private workspace. Having multiple writable copies can be dangerous because they can be accidentally modified. Nestor [Nes86] aptly describe this copy problem:

A well-known software engineering "rule" states that when there are two identical copies of the same file, at least one of them is different.

A common scenario is that a user finds a bug and corrects it in the local copy. Other components that depend on this modification will not work with the old uncorrected version. This may lead to puzzling results when these components are sent to others.

Even when there is only one copy of a version, modifications are frequently undesirable. One of the reasons that old versions are stored is to record historical development. By changing an old version, part of this historical information is lost. This can make it impossible to reconstruct previous states correctly. Furthermore, the user may not even be aware that an old version has been changed. When a bug report comes in, the old versions must be retrieved to locate the bug. If some components of this release have been changed in the meantime, it may not be possible to reproduce the bug exactly as reported by the customer. Furthermore, new bugs may have been introduced by these modifications. Thus, tracing and fixing the bug becomes more complicated if old versions can be changed.

All these problems are due to the fact that old versions can still be modified. Therefore, we require that old versions be immutable, it should not be possible to modify them (but it must be possible to delete them). Instead of modifying an old version, a new version must be created (as a successor of the old one) that incorporates these modifications.

6 Completeness of history recording

History recording must be complete. It must be possible to apply version control to arbitrary subsets of the user's data. This means that all types of components can be subjected to version control. Furthermore, the DCWSS must not only supply version control on the contents of individual components, but also on other structures in the storage system such as relations and types.
When history recording is not complete, certain information is not reproducible at later points in time. This is frequently undesirable because it hinders further development.

In our example, versions of all program modules must be recorded. But this alone is not sufficient, because there are other components that belong intrinsically to this program. Other components that are part of the system are:

- the user documentation (plus pictures)
- the specification of the program
- documents that describe the systems architecture
- the shell scripts that are used to build the program.
- test programs, and test inputs plus outputs
- tool programs, such as text formatters and parser generators
- subroutine libraries that have been used

All of these components are needed to continue development on the recorded version of the product.

Note that some of these components, such as pictures and binaries are not simple texts. The system must also be able to record these binary components.

It is also important to record all other features that might influence the behaviour of the program. For example, new bugs may be introduced by using a new version of the compiler, different compilation options (e.g. optimisation), different versions of libraries, or different test data.

7 Frequent creation of versions

Fine-grained history recording, where new versions are recorded very frequently has many advantages.

The first advantage is the possibility to undo unwanted modifications. Users make mistakes, e.g. accidentally deleting an important component. Another case where an undo feature is desirable is when a user has imported a set of components from another user. The important components may be incompatible with the other components from the receiver.

For example, replacing some modules in a program could introduce severe bugs. In such a case the operation must be undone, but this is only possible if the state prior to the import was recorded.

A second advantage of frequent creation of new versions is that they can aid users in examining their own developments. For example, to answer questions like “what did I change today?”. For a bug-fix it is frequently necessary to make modifications to multiple modules, and a good overview of what has been changed is necessary for the description of the results of the bug-fix. Also when enhancements have been introduced in a program, new bugs may show up under testing. In such a case, it is important to know whether this bug was also present in the previous test run. If this is not the case then the bug must have been caused by the modifications since the previous test run. A good and complete
overview of precisely what has been changed since the previous test is very important to
locate the bug.

But all of this is only possible if the correct versions have indeed been recorded. Often
it can only be decided afterwards whether a given state must be recorded. This can be
alleviated if versions are recorded frequently.

# 8 Support for undoing operations

There are several situations where a user can determine that the most recent modifications
are not desirable. The modification may have been a mistake by the user. For example,
an important component has been deleted, or a bug has been fixed incorrectly. Other
situations arise during explorative programming where a user, when confronted with
multiple alternative decisions, tries several approaches in order to compare them. Yet
another kind of “undesirable” modifications are debugging statements that have been
added in order to trace a certain bug. When the bug has been located these statements
are no longer needed and should be removed. Debugging statements that do not alter the
state of the program but only print out information about its behaviour, could be left in
the program, or be excluded by of use conditional compilation. But frequently they clutter
up the code and furthermore not all languages do allow conditional compilation. In these
cases a user will want to undo the most recent modifications and return to some previous
state.

# 9 Version transparency for tools

Which version of a certain components is used must be invisible for tool programs.

Most tools only use one version of a component. For example, a compiler would not be
able to handle two different versions of the module “IO.c”. If at the same time multiple
versions of a component exist one of these must be selected. The version selection for a
given component should be independent from the tool that is used. Without this, either
tools are more complicated because they must know about version selections, or the user
interface is more complicated because the user must supply the names of the selected
versions to the tool.

# 10 Support for variant management

The system must support variants. Variants are versions of (composite) components that
share common sub-parts. Variants commonly arise in program development where variants
of a program exist. The most common cause for the existence of multiple variants of a
program is the need for the program to be portable between different hardware/software
platforms, but variants may also be needed to handle versions for different countries.

Variants are different from versions that arise through historical developments or parallel
development. Multiple variants coexist at the same time. In general one variant is not more
important than others. In historical developments newer versions in general replace older
ones. In parallel developments the results of different lines of development must finally be
merged to form one version that incorporates the improvements of all lines. With variants
this is not the case, each variant has its own reason for existence and will persist through time.

Different variants share certain parts, while other parts are private to this variant. Updates to common parts must be incorporated in all variants, whereas updates that are specific to one or more variants should have no effect on other variants.

Since the number of possible combinations grows rapidly if there are several modules that can have variants, some form of automated support is needed for the selection of a consistent combination of versions.

1.1.3 Consistency management requirements

11 Consistency

The system should help in maintaining consistent states.

There are certain desirable properties that combinations of components should have. For example, given a set of components that together form the source text of a program, it should be possible to compile this collection. Some collections of components will have this property while others do not. Not all combinations of components are consistent. For example, if module “IO.c” is changed by adding a new procedure to its interface, then new versions of other modules that use this procedure, are not consistent with old versions of “IO.c”.

There are different “degrees” of consistency. A more consistent collection of components satisfies more of the properties we would like it to have. Some consistency requirements are very basic, such as the requirement that a program must be compilable. Another requirement is that the program must be able to pass regression tests. These requirements could be verified automatically. Other inconsistencies are much harder to find. Inconsistencies between the code, the specification, the description of the systems architecture and documentation arise very frequently, but finding and solving these automatically is not feasible.

Automated support for detecting and repairing inconsistencies is desirable. In a loosely-coupled environment the need for support is even bigger than in a centralised situation. This is caused by the fact that communication among users is much more complicated in the loosely-coupled case. Furthermore, because separate development occurs frequently, there is a large possibility that many consistency constraints are violated after merging the results of two different development lines.

Inconsistencies are a fact of life in development situations. Changes generate new inconsistencies that must be solved later on. From a somewhat extreme point of view one could say until the first release of a program the product is always inconsistent because the code of the program does not (yet) agree with the specification. Therefore, the system must have a certain amount of robustness against inconsistencies. It is unacceptable that further work is inhibited unless all inconsistencies have been removed. For example, it must be
possible to work on parts of a program, even though there are inconsistencies in other parts of the program.

12 Derived components
The system should support the automatic derivation of components.
In general it is very hard to solve inconsistencies automatically. However, one important exception are derived components, i.e. components whose contents can be computed entirely from a set of other components. A prime example of a derived component is of course a program that can be produced automatically by invoking the compiler and linker on its source texts. It is in general highly desirable that derived components are consistent with their inputs. Take for example, a programmer who is debugging a program, and has made some changes to the sources. If the program is accidently not recompiled, it still shows the old behaviour even when the bug has been fixed by modifications to the sources. This type of mistakes is highly frustrating.
Nevertheless, when programmers have to keep track of recompilations manually, this kind of errors occur on a regular basis, especially in large complicated programs. One solution would be to recompile all dependent components whenever some of the input components has been changed, however, this may be expensive.
It is tempting to try to automate the administration, and to let the computer do the hard work. An automatic system that handles these derivations is called a derivation manager.

13 Consistent version combinations
For most activities only one version of a given component is needed. When there are multiple versions of several components one version must be selected for each component. The behaviour of the ensemble depends on the specific versions that are combined. When there are many components in the ensemble each of which has multiple versions the number of possible version combinations grows very rapidly. Many version combinations are inconsistent, e.g. they might make a program unacceptable to the compiler. The DCWSS should attempt to avoid inconsistent version selections. The selection mechanism should be such that the default combination is likely to be consistent.

14 Detecting dependencies
Components depend on other components. For example, a documentation component may include certain pictures, or a program module may depend on the interface of another module.
Dependency information can be used in two directions: on which components does this component depend, and which components depend on a given component.
Dependency information is very useful when incorporating work of others, since it is possible to determine which components are affected by this operation.
Tracing and maintaining dependency information is tedious and error-prone. Automated dependency detection is therefore desirable. This is difficult to do in general because
dependencies depend highly on the kind of components involved. Detection of dependencies of a document requires different detection algorithms than detection of dependencies among program modules.

1.1.4 Cooperation requirements

15 Support for parallel development

Parallel development occurs when multiple users are working on versions of the same data. There are several causes for parallel development.

In a loosely-coupled environment parallel development is of course unavoidable, since it is frequently impossible to communicate with another site. It is unacceptable to halt all activities until a connection becomes available.

But even in a tightly-coupled environment parallel development occurs frequently. Users want to do their own work without being disturbed by the activities of others.

Another reason for parallel development is the so-called “bug-fix during development” situation. Development of the product normally continues after a release, thus the current development environment of the developers will be different from the one that was used to produce the release. When bug-reports come in, it is not generally possible to fix this bug in the most recent version of the product. There are several possible obstacles:

- The newest version is not yet in a sufficiently stable state, because it still contains too many bugs.
- The newest version contains new enhancements, for which the user should pay extra.
- The bugs may be due to the special customisations for this customer, which are not part of a general variant under development.
- It may not be possible to reproduce the bug with the current version (this of course does not imply that it has been fixed, it may manifest itself in a different way).

The common solution is to have two lines of development, one that contains the original release plus bug-fixes, and another line that contains the developments for the next release.

Support for parallel development means that there must be mechanisms to handle multiple lines of development. It should be possible to examine the current state of a certain line, and to exchange data among different lines. It is also necessary to coordinate the different development lines in order to prevent duplicate activities and conflicts.

16 Support for exchange

Users regularly want to exchange components with others. When a new version of a component has been developed, other users must be able to obtain a copy. Thus there must be a mechanism that allows the exchange of data among users.
In loosely-coupled environments, exchange between users on different sites is not trivial, due to slow/absent communication lines. Exchange requests cannot always be executed immediately in such an environment, but may have to wait until later.

Users normally exchange groups of related components. For example, when Jack, who is responsible for the documentation, has completed a new version of the manual, all components that are part of the manual must be exchanged with Jill.

Exchange of components between users can be done manually. Users then have to decide themselves when components should be transferred. However, with the manual method it is easy to make mistakes, e.g. forgetting certain components, forgetting to transmit components to a certain user etc. Therefore, it must be possible to automate the exchange process.

It is not always desirable that all users always automatically receive the newest version of a component. It is usually unwise to distribute a new version of a program module until it has been tested. For describing automated exchange it is necessary to describe the conditions when a component can be exchanged.

17 Private workspaces

Users must be provided with private workspaces to do their own developments. The activities of users frequently interfere with one another. For example, if Jack is debugging the program while Jill keeps on modifying it at the same time, it is very difficult to assess the effects of a given modification. The situation is even worse when Jack is not aware of the activities of Jill. If both are editing the same component at the same time, the final result will contain the modifications of the last user that finished editing. The modifications made by the other are lost because they will be overwritten.

Users must be able to isolate themselves from the activities of others, they should be able to have their own workspaces in which they can store their own components.

18 Support for merging

After a certain period of separate development the results of the developments must be merged with one another. The system should support this merging operation.

In order to merge two development lines it must be determined precisely what has been modified in each development line. Potential conflicts may be detected at this stage. For example, two users may have been editing the same component. Some of these conflicts can be resolved automatically if it can be determined that modifications do not interfere. Usually multiple components have been changed, therefore the system should be able to handle merging multiple components. Furthermore, the system must be able to handle merging of all entities in the storage system and thus also be able to merge relations and types.

Different types of components require different merging algorithms. For example, an algorithm for merging program texts should be different from an algorithm for merging pictures.
The system should perform automated merges when this is possible. When conflicts arise that cannot be solved automatically the system must present the conflicts to the user and offer possible solution strategies.

1.1.5 Distribution requirements

19 Robustness against disconnections
In a loosely-coupled environment connections can frequently be absent. It must be possible to continue work when connections fail. Thus the support system should not rely on any centralised services.

One of the consequences of this robustness requirement is that the components that are necessary for a user must be available locally. Waiting for access to components on other systems is not acceptable in a loosely-coupled system.

This situation is very different from the situation in tightly-coupled systems, where it is generally expected that network connections will fail only for short periods of time.

In our example, when Jack is updating the user manual (a version of) all components that belong to the user manual must be available. Furthermore, certain other components are also needed, for example the \TeX-program and the drawing tool that is used for drawing pictures. When the sources or the specifications are needed for updating the manual they must be available too.

20 Work on slow networks
Communication lines in a loosely-coupled environment can be very slow. It is not feasible to simply copy multi-megabyte systems due to long delays. Nevertheless, the support system must be able to function adequately under such conditions.

1.1.6 Miscellaneous requirements

21 Efficient storage mechanisms
The system must store (versions of) components in an efficient way. It is possible to store each version as a separate component. However, this leads to huge storage needs. Efficient storage mechanisms are therefore needed, that minimise the amount of disk space, while providing fast access to the components that are stored by the system.

22 General purpose, customisable
The system must be general purpose and customisable.

Computers are used for all kinds of activities such as maintaining data-bases, writing books, doing graphics design, programming, etc.. However, there are certain similarities when these activities occur in a loosely-coupled environment. For example, mechanisms are needed for exchange and cooperation, history recording etc. We therefore require that the support system should be able to support all such activities.
On the other hand each type of activity will put special demands on its support environment, so that it must also be possible to tailor the system.

23 Mechanisms not policies

The system must support different development policies by supplying mechanisms to support these policies. Policies should not be hard-wired into the system.

There are many different development policies. Which policy is most appropriate depends very much on the local working environment.

In some companies a very rigorous policy is followed where all activities are strictly regulated. There is a strict division of tasks. Usually, a permission system exists that determines who can read/modify a given component. Standard operating procedures regulate many activities. Users are not free to exchange components, but can only use components that have been approved by a central authority. Procedures are in place for handling change requests. A change request is a modification request e.g. to fix a certain bug or to add a certain feature. A change request procedure describes who can issue a request, who can solve it and how the results must be reported, and finally who can approve the resulting modifications. Strictly regulated policies such as these are mainly found in large development groups.

A very different, more anarchistic policy of development is found in other situations, mainly when working in smaller groups. Here the situation is not so strictly regulated. Each user could potentially examine and modify all components. In this environment it is certainly possible that several users work on the same component at the same time. Users can freely exchange components. There is no central authority, users are individually responsible for the results of their developments.

A system that is especially designed for a strictly regulated policy will be seen as a harness when used for the anarchistic policy. Similarly, an anarchistic system will give little support for working in a regulated environment.

There is no universally accepted best policy. Which one is the best depends on the local situation. Therefore, a version management system should be able to support different policies. These should not be hard-wired into the system. However, the system must be supply mechanisms that can be used to express policies.

1.2 Existing systems and practises

This section describes how existing systems and practises deal with the requirements that were stated in the previous section. Some of the system that are described are only relevant for some requirements, but do not attempt to meet all of them. Section 1.3 presents conclusions about the state of the art in this area.
1.2.1 Version management practises

Some requirements can be met without specialised automated support, by using certain cooperation protocols. This section describes some of these protocols.

Copy/Modify/Merge

One of the basic development methods, especially used in the development of software, is the copy/modify/merge approach. When developers want to work on a product they start by making private copies of a standard version of the product, called the base-line. They make their modifications not to the base-line but on these copies. When they are finished the results are merged with the results of other developments. After being tested (integration testing) this integrated version could be used as a new base-line. This approach avoids the interference among different users during their development activities, because their modifications only changes their private version, and not the version of others or the base-line version.

Locking

One of the methods to prevent interference during parallel development is called locking. When multiple user are allowed to modify the same component it is possible that two of these modifications conflict. If no special precautions are taken it is possible that one of these modified versions will overwrite the other one. If it can be detected that there are two different conflicting new versions it is also possible to attempt to merge both versions into one new version that incorporates the changes of both. But merging is a difficult process that frequently requires user intervention.

One of the methods to prevent this is to use locks on specific components. Every component is assigned to at most one person, and that person is the only one who is allowed to modify this component. Although locking avoids certain problems, it cannot prevent all of them. Usually components depend on the contents of other components. For example, the routines in one module depend on routines that are defined in an other module. Changes in a module can force changes in the modules that depend on this module. This can only be done if the dependent modules have been locked as well. Locking can be too restrictive if it is handled strictly. Most groups that use locking also have methods to break a lock. This is needed, when the locker cannot be reached, e.g. due to illness.

Locking is a form of pessimistic concurrency control, it is assumed that independent modifications on the same component will normally lead to conflicts, and must therefore be avoided. Optimistic concurrency control assumes that normally independent modifications will not conflict and will allow multiple independent modifications. However, with optimistic concurrency control it must always be checked afterwards whether there were any conflicts, and conflicting modifications must be merged.
Release management

Versions of a product that are delivered to external customers are often called releases. Many products undergo continuous development, and therefore have many different releases. Releases should be carefully version managed, because they represent very important versions. With large computer programs, releases are often tailored to the demands of a specific customer, in such a case there can be many different releases with different characteristics.

With computer programs, clients often report bugs or want improvements. These are called change requests. In order to handle these change requests in a proper way it is important to record extensive information about the properties of a release. In general, such a change request must use the development environment of the original release to make the modifications. Therefore, it is important to record in a precise fashion the development environment of a specific release.

For bug-fixing it is of crucial importance that the bug is reproducible. Therefore, it must be possible to reconstruct the entire contents of the release. In the case of computer programs, the customer generally receives a binary version of the program, plus some auxiliary files and documentation. These form only a part of the release as it must be recorded by the manufacturer. The manufacturer must also record the sources as part of the release. But there are other parts of the development environment of this release that can be recorded too, such as internal documentation/specifications, and the tools that were used to build this release. Depending on the nature of the product the extend to which the complete development environment is recorded can vary. Many programs depend on the version of the compiler that was used to compile it. Many compilers contain bugs, so the version of a program that has been compiled with a new version of the compiler can show a different behaviour. In general, it is therefore wise to record the version-identification of the compiler that was used with a specific release. Under more strict conditions it is also desirable to store (the version of) the compiler itself as part of the release. For embedded applications where the binaries are put in ROM, it is very important to be able to reproduce a compiled program exactly. In these cases both the produced binary and the compiler will be stored as part of the release.

Compilers only form one class of examples of tools that form part of the development environment of a specific release. Other examples of important tools are parser generators, locally developed filters, and utility scripts. It is desirable that these tools are also stored as part of the release.

Another class of components that form the development environment of a release are descriptive documents. This class contains documents like manuals for the product, and for the tools, specifications and implementation descriptions. These descriptive documents should also be stored as part of the release.
1.2.2 SCCS and RCS

In one of the most primitive methods for version management each version is stored as a separate file, and it has to be recorded which version corresponds to which file. Some operating systems (e.g. VMS) and tools (e.g. Emacs) provide this functionality by storing numbered backups. This scheme has several drawbacks. First of all, storing all versions in their entirety consumes a rather large amount of disk-space. This makes users reluctant to record new versions. Another disadvantage of this scheme is that it is difficult to store information about the versions. Thus it becomes difficult to answer questions like: "what is the most recent version of this item?", "how many versions of this item exist?", "what is the status of this item?".

The earliest version management systems were designed to solve these problems, and became rapidly popular. The first such system was SCCS [Roc75]. Several similar systems, e.g. RCS [Tic85] and CMS [CMS] soon followed.

The structure of these systems is very similar. They are mainly designed to store different versions of one file. All versions of one file are stored together in a data-base, which is normally just another file. The versions are commonly stored using a compression technique called delta-compression. A delta is a description of the difference between two versions. Deltas can also be used to construct one version from the other. The trick is that there is no need to store two complete versions. Instead it is sufficient to store one version plus the delta. Since the delta's are normally very small relative to the size of the entire file, this can reduce the amount of storage considerably.

Delta techniques exist in two varieties called forward and backward deltas. With forward deltas (as used in SCCS) the oldest version is recorded in full and delta's must be applied to construct newer versions. With backward delta's (used in RCS) the most recent version is stored in full, and deltas must be applied to construct older versions. Since often the most recent version will be used more frequently, backwards deltas are somewhat faster.

The versions maintained in these systems cannot be used immediately, before a specific version can be used in an operation, the contents of that version has to be copied to a file, this operation is commonly called "check-out". Similarly, there is an operation, "check-in", to record a new version.

These systems record how versions are related, by maintaining a successor relation between versions. A version X is called a successor of another version Y if X has been produced by modifying Y. The successor relation in these systems has a tree-like shape. Users can create branches by adding new successors. The standard naming structure used to identify versions is based on this successor relation.

These systems have locking: only one user can lock the file. Other users can perform check-out but cannot check-in. There are methods to break a lock.

Comments can be added to describe a version. The system automatically records who checked in the new version, plus the time and date of the check-in.

These systems also supply a status attribute (a string) for each version that can be used to record the status of a version, e.g. experimental, stable or released.
Because the units for modifications in these systems are lines of text, they cannot be used to store versions of binary files. Reichenberger [Rei91] has proposed a technique that can be used for computing deltas on arbitrary files.

1.2.3 Derivation managers

In most computer environments there are certain components that can be derived by a program from other components. An automatic system that handles these derivations is called a derivation manager. This section briefly introduces some derivation managers. Chapter 5 contains a more extensive discussion of this topic.

Make

The first well-known derivation manager was Make [Fel79]. Make has been developed for Unix, where it soon grew very popular.

The basic concept of Make is very simple. Make uses a so-called Makefile that contains for each derived file, a list of files on which it depends, and a command to generate the derived file when it is out-of-date. The algorithm to determine whether a derived file is out-of-date checks if one of the files on which it depends has been modified more recently than the derived file itself. If a derived file has gotten out-of-date Make will invoke the command to re-generate it.

Make is a simple, but effective tool. Nevertheless, there are certain problems associated with it. In the first place, users must still explicitly invoke Make, if they forget to do so, they can still by mistake use out-of-date files. Furthermore, all dependencies must be explicitly specified to Make and it is easy to forget certain dependencies. For example, strictly speaking, the results of the derived files also depend on the Makefile in which they are described because the Makefile normally contains options that are used for compilations etc. However, this dependency is almost always ignored. For certain programming languages tools exist to generate dependencies by examining the source files.

Another practical problem with Make is the lack of transitive dependencies. If we take a program, then normally in the first step the source files are compiled to give relocatable binaries, in the second step these binaries are then linked and loaded to give the executable program. In such a case it is not possible to tell Make that the intermediate relocatable binaries are not necessary. If these are deleted, Make tries to compile them, even though the executable program is newer than all its sources.

Odin

The Odin system [Cle88a, Cle86] is an advanced derivation manager. Odin knows how to convert many file types into each other. Basically, Odin views derived files as the result of applying functions to its inputs. All derived files are stored in a cache that is transparent to the user. Because the cache is entirely maintained by Odin, it is able to detect transitive dependencies. Therefore, it can overcome the problem with intermediate files that Make
has. Odin does not unnecessarily recompile intermediate files, as long as the final derived file is up-to-date with its source files. Therefore, these intermediate files can be deleted from the cache.

Odin assigns a type to each component. Example types are "C" for C files, or "exe" for executable binaries. Type information is used to determine which operations are applicable to a given component.

The tools in the Odin system are described by a derivation graph. This graph contains for each tool a description of its inputs and output with their respective types. Tools that produce multiple components are described as returning one compound derived component. The derivation graph is rather different from a Makefile, in a Makefile dependencies between component instances are described, while Odin puts the emphasis on the tools, and thus describes dependencies between different component types.

1.2.4 Loosely-coupled development of programs

A very interesting example of very loosely-coupled cooperative work can be found in the development and distribution of some non-commercial programs in large networks. Authors of programs make sources available to users, who can make their own improvements to the sources. They can make the improvements available themselves or send them back to the author for incorporation in the next release. This type of development is frequently found in Unix environments. Authors announce the availability of sources on Usenet or on mailing lists. Different communication media exist for the exchange of these sources: ftp, uucp, electronic mail, Usenet, or by sending magnetic media such as tapes and floppies by normal mail.

Because programs usually consist of multiple files, certain mechanisms are used to transfer these as a unit. Archiving programs such as tar, cpio, shar, zoo and arc are used to store multiple files and directories in a single archive file. These files are frequently compressed in order to minimise transmission times. Some archivers (e.g. zoo and arc) already have built-in compression techniques. With other archivers this is handled by a separate program. Not all communication media can transmit binary data. On such media the data must first be converted to a textual format and after transmission converted back to binary (e.g. by using uuencode).

When changes to an existing big program must be distributed, it is customary not to transmit the entire program, but only the differences (that can be computed by the Unix diff program). These differences must then be applied by the receiver to the original version of the program. This could be done manually, or by using the patch program [Wal88]. Patch has a certain amount of flexibility. When the lines that must be modified are not present at the original line numbers, patch attempts to match lines in the neighbourhood to see if they have been moved. When these can be found patch applies the differences to this new location. In this way patch is used to merge the developments of the receiver with the new diffs. This support is limited, patch will not be handle modifications to the same line.
Because every one can make their own changes, many different versions exist. It is not always easy to determine the properties and development history of a specific version. Many programs have been modified by a large group of people. In some cases (e.g., the news-reader) the original authors have stopped supporting a program, and others have taken over this task. It can also happen that users create their own variant of the program and start supporting this variant (e.g., the trn news-reader is based on tr). In some cases (e.g., dvil2ps) this has lead to a proliferation of versions where it was difficult to determine which version was actually used.

1.2.5 NSE

The Network Software Environment (NSE) is a version management system developed by Sun [NSE88, CFH88]. NSE runs on a network of Sun computers using Sun’s Network Filing System (NFS, [Now89]) and Yellow Pages (re-baptised NIS by SUN).

A central component in NSE is the environment. An environment is the focus for development. Each environment contains a set of private versions of the files that are under development. The files that form part of this environment are not shared with other environments, but mechanisms exist to exchange parts of the environment with other environments. The files in an environment can be subject to version management, using a check-in/check-out mechanism that is comparable to SCCS/RCS.

Every environment offers a different view on the Unix file-system. Certain parts of the file-system are private to an environment, and can only be seen inside this environment. Thus given a file-name the file that is designated by this file-name can be different in all environments. These file-system views are implemented using the Translucent File System [Hen88].

The elements in an environment can be organised by using components. Components are organised in a hierarchical fashion, and are similar to, but independent from, the normal Unix directory structure. A component can contain other components, files and Make targets. Derivations are described by a Makefile, and targets in the Makefile can be represented in the component structure.

The acquire operation is used to obtain components from other environments. This command copies the contents of a component in a recursive fashion. After modifications the reconcile operation merges the component back into the original environment. Conflicts between the version of the component in the original environment and the new version are noted. NSE provides a tool to merge two versions of a file, that is invoked automatically whenever conflicts between these versions are detected.

The standard method of working in NSE is the copy/modify/merge cycle that is described in section 1.2.1. One environment can be used to hold the base-line versions. Different groups of developers each have their own environments. Each group starts by acquiring all needed components from the base-line environment. Following that they all perform their own activities without interfering with one another. Once a group is finished with its activities, their environment is reconciled with the base-line environment.
NSE contains a graphical user interface. In this interface environments and components are represented graphically. After clicking on such a graphical object a menu is presented with the possible operations on this object. Many common operations in the environment can be executed using these menus, such as creating and deleting environments, check-in/check-out of files, building a Make target, and navigating through the component structure. For all these operations, there is a command line equivalent.

1.2.6 DSEE

The Domain Software Engineering Environment [LC84, DSE86] contains several interesting version management facilities. DSEE runs on HP/Apollo systems on a local area network.

Under DSEE, version management has been integrated with the file system. As we saw above, with systems like RCS and SCCS an explicit check-out operation is necessary before the user is able to work with a specific version of a file. Under DSEE this is no longer necessary. There is a certain class of files that are versioned. When user programs attempt to read this file, they can transparently read old versions of this file. Internally, old versions of the file are stored using delta-compression. The user can determine per process which version it should use for each file.

Furthermore, DSEE contains a powerful facility for building derived components, called the System Modeller. A system model describes the general structure of the derivations in one software system. This description only refers to files in an abstract way. It is still necessary to select a specific version of this file. Selection of versions is accomplished by a configuration thread. A configuration thread contains a certain degree of flexibility, e.g. the user can decide to select the most recent version. Which version is selected depends on the time at which the system with this configuration thread is evaluated. One result of this process is a list of which version was selected for which file. This list is called a bound configuration thread.

DSEE maintains an internal cache of all derived values. Furthermore, DSEE is able to distribute the computations that are needed for a system build over multiple machines in the network.

1.2.7 Adele

Adele [BE86, EGK84] is a configuration management system for software development. Adele contains facilities for storing versions of files and controlling their access. Objects in Adele have attributes. Some attributes are system defined but new attributes can be added. Program parts are called modules. A module has an interface, that describes what features it needs and what features it offers, and a body, that provides an implementation of the features that are offered by the module. Interfaces and bodies can have versions. Each version of the interface can have several bodies associated with it that contain different realizations of this interface.
Adele contains facilities for building programs. For a given compilation the correct version of all modules must be selected. Adele allows the user to select versions by using queries on the values of the attributes. Normally, only one version of each module is selected even when the same module is used by several other modules.

Modifying one object can have effects on other objects. For example, changing the interface can influence the body, and a change in the specification of a program will have an influence on the implementation. Adele uses such information to determine the side-effects that a certain modification will have. Such side-effects can be detected at different points: before the modification is stored, after it has been stored, when the affected object is used and on demand after explicit user requests. The user can determine what actions must be performed when a certain (potential) side-effect is detected.

1.2.8 Using file-systems

Traditional support environments use the file-system as storage system. However, the modelling capabilities of a file-system are very limited. In this section we show some examples of important information about components and examine how these could be represented in a file-system. Following sections describe other kinds of repositories.

Normally there is a lot of information about a specific component that is only represented in an indirect way in a file-system, e.g.:

1. Who created this component?
2. Who has made the most recent modification?
3. What is the development history?
4. What is the contents of this component? (e.g. for a program text, which programming language does it contain)
5. Of what larger unit(s) is this component a part? (for example, a program will contain several source components)
6. On which other components does this component depend? (e.g. include-files)
7. With what other components is this component related? (e.g. what is the documentation for this source text)

One solution is to store such information in the contents of the file. In program sources it is common to record this type of information inside comments. One disadvantage of this approach is that there are a large number of different comment conventions. This makes it difficult to extract and manipulate the information automatically. A further disadvantage is that this approach cannot be applied to all files. Certain files have a fixed internal format and it is not easily possible to add extra information. Examples are program binaries, pictures, statistical data sets, data-base files. In these cases it is possible to record this information in some external file but then it must be assured that the original file and the description file must be updated simultaneously. Thus if the file is renamed or transported to a different site the description file must be treated in the same manner.
Some file-systems have extensions that aid in storing information about the properties of the components. The files in the Apple MacIntosh have resources that can be used to record properties of the file.

Another mechanism that is used to store information about a file is to use its file-name. It is customary in many operating systems to denote the programming language of a specific source text with a suffix. A file-name ending in ".e" would name a C source, and a "tex" suffix indicates a \TeX source. However, this is only a matter of conventions. Because these suffixes are short, it is possible that clashes occur, for example, the suffix .pl is used for both Prolog and Perl programs.

Naming structures are frequently used to group related files together, e.g. in a common directory. This mechanism is rather limited. Under most versions of Unix the compiled version of a program and its manual page and its sources are stored in three different directories, something like /usr/local/bin, /usr/man/man1, and /usr/local/src. This does not make it easy to locate the sources for a program once it is known where the binary is located.

In traditional file-systems it is difficult to maintain references between files. Usually references to other files are denoted by a relative or absolute path-name. This mechanism breaks down when a file is renamed or moved to a different directory, since it is impossible to find all references to this file.

1.2.9 Object Management Systems

Older version management systems are based on file-systems. As we have discussed before file-systems do not form an adequate basis for a repository due to insufficient expressiveness. Many of the newer engineering support environments are not based on file-systems, but on a specialised type of data-bases called Object Management System (OMS). Examples of such systems are PCTE (see below) [ECM90], Orion [KBC+87], Isis [F+88], Damokles [DGL86], Postgres [Se87], Gemstone[BM+89], and Emerald[BJ+87].

An OMS provides the notion of (typed) objects that are stored persistently. Objects are typed, the type of an object determines its internal structure and the operations that can be performed on it. Objects have a number of attributes (also known as instance variables). Furthermore, an OMS contains some mechanism to store references among objects.

A special class of OMSes are the object-oriented data-bases [KL89, ZM90, Bro91]. Object-oriented data-bases have been influenced by developments in object-oriented languages (of which Smalltalk [GR83] is the prototypical example) and traditional data-bases. Object-oriented data-bases are a relatively new concept and there is no perfect consensus about what features they should possess. However, most systems have at least the following properties: Objects are typed, the type of an object is frequently called a class. Classes are organised in a class hierarchy that forms a partial ordering. A class has a set of operations (frequently called methods) associated with it. New operations can be defined on a class. A class can inherit operations and attributes from its super-classes. Inheritance leads to a special kind of polymorphism, which method is executed depends on the class in which
the method was found. Two different forms of inheritance exist: single inheritance where each class can have only one immediate super-class, and multiple inheritance where a class can have multiple immediate super-classes.

When persistent objects become long lived, the possibility that their types must be changed becomes real. This phenomenon is called type evolution. Some work has been done on this topic, see [Zdo86, SZ87, BCG*87]. Certain type changes are “upwards-compatible” such as the addition of new methods and attributes. Type changes that are not upwards-compatible cause difficult problems. Banerjee [BCG*87] contains a taxonomy of type changes for Orion and the way that these are handled. Changing all instances of a given type immediately can be very expensive. Zdonik [Zdo86] solves this problem by letting the programmer defining filters that convert objects from one type to the other. However, type evolution is still largely an unsolved problem that is ignored in most OMSes.

1.2.10 Historical data-bases

Historical data-bases (e.g. see [CW83, Se87]) are data-bases that also record the development of the data they contain. Such systems are typically able to answers questions about situations in the past, e.g. “give me the number of employees on January 1st 1991”. In this respect they are very similar to standard version management systems that are also able to store information about historical states. What differentiates historical data-bases from standard version management systems is their ability to answer queries about time ranges. They can answer queries like: “what is the average number of employees over the past 3 months”, or “what is the lowest salary that has been paid over the past 4 years”.

Postgres [Se87] is an example of such a data-base. Postgres is a successor to the Ingres relational data-base. In Postgres every tuple in the data-base is tagged with a time-stamp. Thus Postgres is able to answer the queries that were mentioned above. In Postgres the log of operations that were performed on the data-base can be treated as part of the data-base. This make it easy to do recovery after crashes.

Historical data-bases have not been used for supporting cooperative work.

1.2.11 PCTE

The PCTE Portable Common Tool Environment [PCT] is a distributed object management system. It provides typed coarse grained objects, with inheritance between types. PCTE also provides typed relationships (called links) between objects. Links have been designed in such a way that they act like a superset of the Unix file-naming structures. The types of objects and relations are described in a schema definition set.

PCTE contains composite objects. A composite object consists of a graph of objects that have been connected with a special kind of links, that are called composition links. Composite objects can share sub-parts.

The PACT Version Management Common Services [PAC89], provides basic version management facilities on top of PCTE. Versions of a component are stored as different
objects. Successive versions are related by a pre-defined successor relation. VMCS provides a call to create a version of an object. The current state of an object is be stored in a new object, that is related to the object via the successor relation. VMCS is also able to store versions of composite objects.

VMCM [BLM91] is an implementation of Adele (see section 1.2.7) on top of VMCS.

1.2.12 EPOS

Epos [LCD'89] is an object management system that has been specifically designed for software engineering applications. The versioning model in EPOS is based on the notion of functional changes. A functional change is a modification to one or several objects. For example, all modifications that were needed to add a new feature to a program would be described as one functional change. Every functional change is associated with an option: a boolean variable that can be true or false. A user selects which functional changes must be visible by selecting a choice that sets the value for each option. Functional changes are global to the entire data-base. All modification tasks have an ambition associated with them that determines in which version choices the modifications will become visible.

A choice can be made stable, this means that the state of the data-base under this choice becomes immutable.

1.2.13 KBSA

The KBSA (Knowledge Based Software Assistant) Framework described in [KL91], provides version management facilities in a fine-grained object management system. Objects have named slots that can contain pointers to other objects.

Related objects are grouped in Epochs. Versions are recorded of an entire epoch. Thus references among objects are also version-controlled. Epochs are described by a configuration schema that can be seen as the type of an epoch. A configuration schema includes:

- A structure description of the objects in the Epoch.
- An interface description that describes its relation with other Epochs.
- A verification description that describes how it can be determined whether an Epoch is consistent.
- Version cursors, that describe dynamic references between Epochs.

Epochs are stored by using a delta technique that records modifications to individual slots. The performance of a prototype implementation compared favourably to a previous implementation that stored versions without delta-compression.

1.2.14 The Workshop System

The Workshop System [Cle88b, Cle89] is a version management system for fine-grained objects, such as individual functions or type definitions. The system contains a object
management system. Objects are typed and have a set of attributes, i.e. name value pairs. Every object has a class, and classes are organised in a class hierarchy with multiple inheritance.

The Workshop System contains a special process programming language called SE-KRL, that is a rule-based declarative language. This language can be used to automate certain operations.

The Workshop System is a distributed system. Objects are stored in a global data-base, and down-loaded to individual workstations. This process can be automated by using SE-KRL rules. For example, it is possible to specify a list of objects that must be down-loaded to the workstation, whenever a specific object is down-loaded.

Programmers are working on specific tasks, called “Jobs” in the Workshop System. For every job it is recorded what software objects are associated with it, e.g. what are the new versions that have been created in this job. Jobs function as a working context, when a user wants to continue on a specific job, the objects that are part of this job can be down-loaded automatically from the global data-base. Jobs are also used to identify historical versions, and this can be used to give a more descriptive label for a specific version.

1.3 Conclusions

We now examine how well existing systems meet the requirements that were formulated in section 1.1.

1.3.1 Expressive data storage

Many of the older systems are based on file-systems. File-systems are limited because they don’t supply mechanisms to represent the types and properties of the components and their relations. Many systems attempt to alleviate this problem by building additional mechanisms on top of the file-system.

In file-based systems the directory mechanism is the only available structuring mechanism. This is very limited because it only provides a method to structure components in one tree. Thus it cannot be used to effectively describe products that share common sub-parts. It can also not be used to handle multiple views on the product, e.g. both a view that describes all the documentation of a product and a view that shows each program in the product together with its documentation.

The NSE “components” are more flexible in that they can be structured in arbitrary non cyclical graphs. NSE components are used for many operations on sets of objects, e.g. to record a new version of all entities in a component or for exchanging entire components with other users. The NSE component mechanism is very different from the directory structures and they are not very well integrated. For example, when a file is moved to a different directory, the components are not updated.

Object management systems are much more flexible in this respect because they allow the definition of new naming structures.
As we have seen several times the conventional file-system is not sufficiently expressive, and its inadequacies are often remedied in an ad hoc way. Object management systems that allow the definition of types and relations form the basis of many of the newer version management systems, such as the Workshop System and VMCM.

**Identifiability of components**

All systems provide mechanism to identify components. In file-based system file-names are generally used for this purpose. This mechanism is limited in that there is in general little support for renaming files. This makes it difficult to move files to a different directory. In object management systems object-identifiers can be used to identify a component, this can be advantageous because object identifiers are generally independent from the naming structures that are used to access the object.

All systems have problems to maintain the identity of objects in a loosely-coupled environment.

**Identifiability of versions**

One of the common mechanisms for identifying a version of source texts is to include the version-identification in a comment. The version management system updates this comment whenever a file is checked-out. This mechanism cannot be applied to components that have a fixed internal structure, such as word processor documents or pictures. Furthermore, if the author forgets to add this comment, it becomes very difficult to determine the version of this component. This approach also makes it difficult to write programs that can extract the version information of a given component due to the wide range of comment conventions that exist.

Systems that have attributes and relations, such as Adéle and most OMSes make it easier to handle version information. Relations can be used to record the “is-version-of” relation, and attributes can be used to record the specific properties of a version. Furthermore, if versions are represented as individual objects it becomes easier to distinguish different versions. In file-based systems all versions of a file usually have the same file-name, thus it is not immediately clear which version a file contains.

**Trace-ability of versions and components**

Even very primitive version control systems provide simple mechanisms to record the history and properties of versions. When a new version is recorded it is possible to attach a comment to this version that can be used to describe it. However, these comments are written in natural language and therefore they cannot easily be used for automated processing. Many systems allow a limited number of attributes that can be used for automated processing. Examples of such attributes are the creation date, and the identity of the creator. Some systems (e.g. RCS and DSEE) allow user to identify specific versions with a string (e.g. “Release 4”). These attributes can be used to retrieve a specific version.
One version can normally have multiple successors. NSE is an exception in that branching is not allowed within an environment. In NSE different lines of development must be represented by different environments. Most other systems do allow branches, and record the successor relationship between versions.

Tracing versions of a component becomes more difficult in a loosely coupled environment. All systems depend on a centralised repository. These systems have only limited use in a loosely-coupled environment. It is possible to check-out a component from the repository and modify it on a loosely-coupled computer. However, each new version must be registered centrally. Thus it is not possible to create multiple new versions on multiple places at the same time, because there are no tools available to merge these development histories.

Immutability

Most systems provide a form of immutability, for historical versions of a component. SCCS and RCS for example, make a file read-only at check-out unless the user has explicitly requested a modifiable copy. Immutability is usually restricted to individual files. It is usually not possible to retrieve the previous state of a directory.

Some object management systems, e.g. PCTE, also provide immutability of relations.

Completeness of history recording

All relevant components that make up a product must be recorded. A common procedure is to collect all components for a product into one directory, and its sub-directories. Version management is then applied to all text files in this directory. These tools do not give much support for recording the history of a complete product. First of all, most version management systems can only handle text files, and thus versions of binary information that is part of the product cannot be recorded by such systems. Only a few systems can handle versions of binary information, see e.g. [Rei91]. Second, a product normally depends on certain components that can be shared among products, e.g. compilers, libraries, privately developed shell scripts etc. Such components are frequently not recorded, and version-control systems also do not give warnings when important components are not brought under version-control. Some tools exist that help determining which components are part of a given product. For example, there are tools that can determine which source files were used to create a given program. However, these tools are normally very language specific, and cannot easily be extended.

Important information that is mutable should be subject to version management. However, in most systems this is difficult, or impossible. One example is directory information that is in general not subject to version control, and most file-based systems do not give any support for renaming files and/or directories. Furthermore, there are several other types of information that cannot be subject to version control such as environment variables.

Some systems, especially OMS-based systems, do attempt to provide a more complete recording of history. In these systems, e.g. KBSA [KL91], VMCM [BLM91] version
control is applied to a group of objects and their references. The Epos system [LCD+89] applies history recording to the entire OMS.

Frequent creation of versions
One important barrier against frequent creation of new versions is the speed of this operation. Normally an old and the current version must be compared to determine whether they have been changed, and, if this is the case, how they have been changed. With a large number of components this can be very expensive and will force the user to wait.

Another barrier against the frequent creation of versions is the amount of storage that is used by the versions.

Versions are normally recorded only after explicit requests by the user. It would be better if the system itself could determine when a new version must be recorded.

Support for undoing operations
All version management systems of course provide a limited undo mechanism for individual components. When an incorrect modification is made it is possible to retrieve a previous version of the component and to continue working on the old version.

This only works as an undo operation on components that are subject to version management. Many systems only provide version management on text files, therefore operations on other types of components cannot be undone.

The effectiveness of this mechanism depends on the time granularity for recording new versions. If new versions are recorded very frequently, this undo mechanism is much more effective.

Version transparency for tools
Old versions cannot normally be accessed by tools in a transparent way. Normally one must check-out all versions to a separate workspace and then invoke the tool. The OMSes have an advantage if versions are represented as separate objects, because such a check-out operation is not necessary. Another system worth mentioning in this context is DSEE. In DSEE it is possible to invoke a shell that transparently reads old versions. For each file it can be specified which version must be used. All programs in the shell then see this version if they attempt to read the file.

Support for variant management
Different mechanisms exist for handling variants. One mechanism for handling variants is conditional compilation, parts of a source text can be omitted or included for compilation depending on certain switches. Conditional compilation is often coupled with other macro facilities, e.g. the C preprocessor. Using a language independent macro preprocessor has the advantage that the same preprocessor can be used for different languages. A
Conclusions

The disadvantage of language independent macro facilities is that it is impossible to guarantee that all (combinations of) variants will lead to syntactically correct programs.

With conditional compilation multiple variants are represented within one file. A different approach is to represent each variant as a separate component (see e.g. [GMS89]). This has the advantage that variants can be represented independent from the semantics of the components. Otherwise the system would have to know the tools (e.g. which preprocessor) that must be used to obtain a specific variant. A disadvantage of representing this approach is that the number of different variants can be very high.

Facilities for handling variants are normally incorporated into the system model that describes derivations. When attributes can be attached to components, these can normally also be used to select specific variants. DSEE and Adele are examples of systems that provide good facilities, that appear to be sufficient in most practical circumstances.

Support for parallel development

The common method to handle parallel development is to use locking: only one person is allowed to create new versions of a component at any given time. Locking is not very appropriate in a loosely-coupled environment because network connection problems can make communication with the lock owner impossible for a long period. Locking restricts the amount of parallelism during development, since users can be forced to wait for a given lock.

The copy/modify/merge approach seems more suited for use in a loosely-coupled environment. NSE is a good example of a system that supports this approach. In NSE each line of development is represented as an environment. When a new line of development is created a new environment is made that becomes a child of the original environment. When development in the child is completed, it is merged back with the parent environment. At this stage NSE can detect conflicting updates. A disadvantage of NSE is that all version information about development within an environment is lost when an environment is merged with a parent.

Support for exchange

Some support for exchanging individual files is offered by all systems. Most of these exchange methods rely on the existence of a centralised repository. For example, it is not easily possible to exchange a version of a file that is stored in RCS/SCCS, because for a true exchange the RCS/SCCS data-base file (in which all versions are recorded) should be copied, too. The original version of the data-base file must then be deleted, because RCS/SCCS assume that there is only one data-base file for each version-controlled file, and they cannot merge two RCS/SCCS data-base files.

The normal procedure is to have one centralised repository in which all versions are stored. All exchanges go via this repository. One user has to check-in a version into the repository, and the second user can check-out this version from the repository.
Support for exchanging multiple components is rather weak in most systems. In file-based systems an easy way to exchange multiple components is to copy entire directories. Frequently only some of the files of several directories must be copied, and then each file must be specified individually. The situation becomes even more complicated when the files are subject to version control. Normally, when a user wants to check-out versions of multiple components from a repository, the desired versions must be specified for each file. Some systems allow user-defined labels for identifying versions (e.g. "Release4"). If the same label is used to identify a version of several files, this label can be used, e.g. to retrieve all files of "Release4". However, using labels has the disadvantage that it may be necessary to declare new labels for an exchange. A large number of labels will then only be used once. Having a large number of labels is also confusing.

NSE offers better support for exchanging a set of related components. The NSE "components" allow organising the data in the workspace. NSE components form a directed acyclic graph, and this can be used to define multiple ways of grouping data. For example, one NSE component could contain all files that were related to a specific program (including its documentation) while another NSE component could contain all documentation files for all programs. The documentation files would then be present in two different NSE components. NSE components are used for exchange between users, when a component is exchanged all its sub-parts are automatically copied. Thus NSE components offer a structured method for exchanging a group of related components.

In object management systems users can use queries to define the set of versions with which they want to work. This is a very general and flexible approach.

Private workspaces

A common scenario for providing private workspaces is to let users perform developments in their own private directories. Files are copied from a central repository or from other directories. When developments are finished the files are copied back. With this approach users are effectively isolated from the possibly harmful activities of others.

Many systems provide little support for transmitting multiple components between directories or to and from a central repository. Determining which components must be transmitted is usually done manually, and it is very easy to overlook some components. This is especially true when a user has renamed a component, created a new one, or has unexpectedly modified a component that was checked-out read-only. Most systems do not detect such conflicts. A better approach is found in NSE where entire NSE components are transmitted between environments. NSE detects creation of new components and renaming of existing ones. However, NSE cannot detect the creation of a new file, unless a new component was explicitly created to represent it.

Support for merging

Version management systems offer limited support for merging the results of parallel development. NSE is one of the more advanced systems in that it can compare the
contents of two environments and offers some assistance in merging source files, but a high degree of user intervention is still needed in this process. Chapter 7 that describes the merging facilities in CAMERA contains a more detailed description of existing merge tools.

Consistency
Many systems provide ad hoc mechanisms for achieving consistency. One example of such a mechanism are locks that prevent updates by multiple users at the same time. Other mechanisms are the maintenance of base-lines, automated derivation management etc. None of the systems provide general mechanisms that allow users to specify arbitrary consistency predicates and to give warnings when these predicates are violated. Some systems, e.g. NSE, DSEE, and especially Adele, provide trigger mechanisms that could be used to (partially) implement such a mechanism.

Derived components
The model that is used in the Odin system, and to a somewhat lesser extent in DSEE, appears to be a good one. Derivations are modelled as functions, and Odin attempts to control all factors that influence the result. This makes the results of derivations very reproducible. Because Odin uses a type based approach the description of the derivations normally need not be provided by the user. An internal cache stores the results of a derivation, so they can be shared among several users. Derived components that have not been used for a long period are automatically deleted from the cache, thereby freeing up disk-space.

The derivation management in CAMERA is described in chapter 5.

Consistent version selections
Most configuration management systems provide mechanisms for selecting versions. Both DSEE and Adele have elaborate version selection systems, based on the attributes of components. A disadvantage of these systems is that the correct values of an attribute must be specified for each component. Both systems appear to work well under practical circumstances, even for large software systems.

Detecting dependencies
Automatic detection of dependencies is limited to derivation dependencies. Several systems provide automatic detection of these dependencies. For example, there are several tools that can automatically generate a Makefile for a given set of C source files. Also, certain variants of Make maintain a .makestate file that registers which files have actually been read for the last time that a certain Make step was executed. This is normally implemented by using a modified compiler. Odin contains tools that parse program sources to determine the dependencies.
All of these tools are highly dependent on the particular details of the programming languages involved.

**Robustness against disconnections**

All version control systems that were mentioned above can only work effectively on tightly-coupled networks. Most distributed systems have certain mechanisms for handling short disconnections. Some have special built-in mechanisms (the DSEE store and forward manager) others rely on the operating system to handle this (e.g. NSE). Even systems that are conceptually loosely-coupled, such as NSE and the Workshop System, have an implementation that requires frequent access to global services.

**Work on slow networks**

Because virtually all version management systems require frequent access to central repositories, they give a poor performance on slow networks. None of the systems attempts to reduce the amount of network traffic in an intelligent way, e.g. by sending deltas instead of entire versions.

**Efficient storage mechanisms**

Many systems use delta techniques for storing versions of text files. These techniques are normally very effective in reducing the amount of storage. Access-times for access to old versions by these techniques are usually acceptable. In DSEE delta storage has been implemented in the file-system itself and access-times to version-controlled files are similar to those of non-version-controlled files.

These techniques are not applied to non-text files (except for [Rei91]). Thus systems either cannot handle version control on such files, or they must store the entire copy.

**General purpose, customisable**

Virtually all version control systems are targeted towards support for software development. Most systems are only able to support version control on text files, and not on arbitrary binary files. File-based systems do not allow the handling of attributes in an elegant way.

File-based systems generally support a limited number of types, e.g. many systems implicitly distinguish text files and binaries. New types cannot easily be handled. Object management systems are generally much more flexible.

**Mechanisms not policies**

Many systems attempt to enforce a specific policy. Virtually all systems by default apply locking, thereby allowing only one user to edit a given component at the times.
Few systems provide mechanism that help defining other policies. DSEE provides a notion of tasks, and makes it possible to define what operation must be automatically performed when the tasks is completed. Another mechanism can be found in NSE where notifications can be attached to components. The notification contains commands that are executed when the component is modified. These mechanisms are useful but can only be activated under a very limited number of situations.

A notable exception is the Workshop System [Cle88b]. It has rule-based language, SE-KRL. Rules have a pattern that describes the conditions under which the rule will be activated. There are several kinds of rules, e.g. rules to describe the communication between the global data-base to the local workshop process, rules that describe operations that are executed automatically, and rules that describe illegal situations. SE-KRL is a flexible language that can be used to program a wide range of policies.

**Summary**

The main conclusions about the suitability of existing systems with respect to our requirements can be summarised as follows.

No systems can function adequately in a loosely-coupled environment. The data model that is used for most system (basically no more than a file-system) is not sufficiently expressive. Version management systems that are based on object management systems are attractive but few of such systems exist. Version management systems usually cannot apply history recording for all relevant components in a product, but only some. New historical versions are created infrequently, this makes it difficult to support undo operations and gives only a very coarse view of the actual historical developments. Mechanisms for exchanging data among users are generally primitive.

The following chapters show how CAMERA attempts to alleviate these problems.
2 The anatomy of CAMERA

The previous chapter introduced the requirements for a version management system for use in loosely-coupled distributed environments. This chapter gives an introduction to the global architecture of the CAMERA system. CAMERA belongs to the group of object-oriented data-bases, section 2.1 introduces the main features of the OMS model. The following section (2.2) lists some of the problems that are encountered when version control is applied to OMSes in a loosely-coupled environment. Section 2.3 describes snapshots and describe how they can solve some of these problems. Roughly a snapshot is a complete set of versions of all objects that are part of a user’s workspace at a certain point in time. Individual workspaces are represented by environments that are introduced in section 2.4. CAMERA contains two different type of OMSes: the Snapshot OMS and the Album OMS. The Snapshot OMS is used for the normal workspaces (environments) of the user. Thus the Snapshot OMS is used to store the products that are being developed. Its contents are subject to version control and are recorded in the form of snapshots. The second type of OMS is the Album OMS that is described in section 2.5. The main function of the Album OMS is to record data about the development process. Entities such as environments and snapshots are represented as objects in the Album OMS. The contents of the Album OMS are not subject to version control. In a system for cooperative distributed work facilities are needed for the exchange of data among the participants, that are introduced in section 2.6. CAMERA contains some special features because it was developed for use in loosely-coupled distributed networks. The final section (2.7) of this chapter gives an impression how CAMERA can be used. The following chapters describe important detail aspects that have been introduced here.

2.1 The OMS

In the previous chapter we have argued that traditional file-systems are not sufficiently expressive. They do not offer adequate facilities for describing and relating complex data. Object Management Systems offer better capabilities in this respect. Therefore, it was decided to include an OMS in CAMERA as underlying repository. Actually there are two slightly different types of OMSes in CAMERA: the Snapshot OMS and the Album OMS (see the following sections). However, the basic data model of both is very similar and in general we will use the term OMS when describing common features. Some special features of the Album OMS are described in section 2.5. The remainder of this section
contains a short overview of the main features of the OMS model, it is described in detail in chapter 3.

The CAMERA OMS belongs to the group of object-oriented data-bases. Objects are typed, each object belongs to a specific class. The class of an object determines its internal structure and the operations that can be performed on it. Operations are described by methods. The OMS contains relations that can be used to store relationships among objects. Relations can be used to structure the contents of the OMS, e.g. to group related objects. A special type of relations are the computed relations. The value of a computed relation is computed by special functions, called recipes.

The OMS is extensible. New objects, classes, relations and methods can be added by the user. Classes and relations are represented themselves as objects in the OMS. New classes and relations are created by sending messages to the appropriate meta-objects. Therefore, there is no need for a separate data definition language.

The OMS is a general purpose system that is not tailored towards a specific application. It can therefore be used to model a wide range of applications. For example, the OMS can be used to mimic a standard Unix file-system, by declaring the appropriate classes and relations. This property is used to allow the usage of standard Unix tools on the CAMERA OMS (see chapter 8).

Simpler OMSes, such as PCTE [ECM90], only offer facilities to describe and manipulate passive data, they do not provide facilities to add new operations. The CAMERA OMS does offer this extensibility since new methods can be added to the OMS, thereby enhancing its functionality.

The OMS contains facilities for handling derived values which will be described in chapter 5. This so-called derivation management will guarantee that derived values are always consistent with all of their inputs. The derivation management automatically detects input dependencies of derived values. This information is updated automatically and is made available to the user.

Another feature of the OMS that aids in maintaining consistency are triggers. Triggers contain operations that are automatically executed when certain conditions arise. Triggers can be used to warn the user when certain inconsistencies are detected or they can try to solve them automatically.

### 2.2 Version control in OMSes

A fundamental notion in CAMERA are the snapshots that will be described in section 2.3. Snapshots form our solution to some of the main problems that are associated with version control on multiple objects and problems that are caused by distribution in a loosely-coupled environment. This section describes these problems, and the next section describes the notion of snapshots.
2.2.1 Version control on multiple objects

A problem that must be solved when applying version control is how version management of multiple objects is handled. This problem is not unique to OMSes, similar problems arise in traditional file-systems if multiple files are version-controlled. However, the latter do not normally offer much assistance in solving them. One of the advantages of using an OMS is that these problems are made explicit, and that an OMS can more easily offer support for (partial) solutions.

Users will manipulate multiple components each of which is version controlled. Normally, only one version of each version-controlled entity is used. When the number of version-controlled entities and/or the number of versions per entity grows, the number of possible version combinations will increase rapidly. However, many of these combinations are not consistent. For example, modifications to one module of a program can destroy its backwards compatibility. Some versions of other modules can then no longer be used together with this new version. In a similar way newly developed versions of such a module may not function with old versions of our first module.

One of the ways that an OMS can help the user in making consistent combinations is by applying version control to composite objects. In this approach a set of related objects, and their mutual references is treated as one composite object. When a new version is created of one of the objects in the composite object (or of its references), a new version is created of the entire composite object. The assumption here is that each version of the composite object is internally consistent, which is very likely since each version of a composite object represents a set of versions of its member objects that have actually been used together. Inconsistent versions of a composite object can be deleted. Many OMSes supply version control on some form of composite objects, see for example [MSK89, CFH88, PAC89, HK88, SZ87, LCD+89].

Versioned composite objects can avoid many inconsistency problems. But this approach still has a number of disadvantages.

Every time a new version of one of the component objects is created, a new version of the entire composite object must be created, too. In many systems, especially the object-oriented systems that use instance variables (pointers) as reference mechanism, this implies that new versions must be created of all objects in the composite object that refer to the new version of the changed component. This can be expensive in terms of storage space.

Different versions of a composite object can share versions of some components, which can lead to very complicated structures. This will complicate some operations on the OMS. A version of an object can only be safely deleted if it does not occur in any version of a composite object that is still being used. The complexity of these structures will make backup and restore operations more difficult.

Another problem is the selection of the objects must be included in a composite object. If we want to make a composite object that contains a program it seems logical to include all of its sources in one composite object. Many programs depend on certain standard
library files, e.g. under Unix virtually all programs depend on standard include-files. Must these standard files be included in the composite object or not? Must the specification and documentation of the program also be included or not? Similarly, compiled programs depend on tools such as a compiler or parser generator. Should these tools be part of the composite object or not? Thus it is not always easy to determine how a composite object must be constructed.

Having composite objects still leaves open the problem how to apply version control to multiple composite objects. The easy solution is to include these composite objects together in one bigger composite object but this can lead to very large composite objects. Version control on composite objects is in general not transparent for all tools. Normally the tool must know on which specific version of a composite object it must operate, and in many cases operations on versioned objects will be different from similar operations on non-versioned objects.

2.2.2 Problems with distributed version-controlled OMSes

Several OMSes (e.g., PCTE [PCT] and Emerald [BHJ+87]) have been designed for use in a distributed environment. But these systems can only function satisfactorily in a tightly-coupled distributed environment. In a loosely-coupled environment new problems arise.

One of the main problems is guaranteeing that all objects that are needed are indeed available on a given node in the network. Because the network in a loosely-coupled environment cannot be trusted (a copy of) all needed objects must be locally available. In large and complicated systems determining which objects are actually needed, can be quite difficult.

The design of tools is more difficult for a loosely-coupled environment. References may refer to objects that cannot currently be accessed, all tools must be prepared to check for the availability of objects and must contain exception handling code for when these objects are not reachable.

The propagation of changes also is more complicated in loosely-coupled situation. Modifications on one site may need to be propagated to other sites, and it can be difficult to determine which modifications must be propagated to which site.

It is very difficult to have objects that are shared by all sites because these objects may not always be reachable. It is possible to have multiple copies of these objects, but in such a case the value of a copy at one site can be different from the copy at another site. Furthermore, some protocol must exist to determine who can modify such global objects.

2.3 Snapshots

In the previous section we have seen some of the problems of version control in an OMS, and the problems in a loosely-coupled environment. These problems have had an important influence on the design of CAMERA.
The versioning model of **CAMERA** is based on what we call *Integral Version Management*: version management is applied to an *entire* OMS. Thus version management is applied to all objects and their mutual references at the same time. This is different from the approaches discussed earlier where version control is only applied to parts of the OMS. Integral version management treats an entire OMS as (a version of) one composite object.

A **snapshot** is one state of an OMS, called the Snapshot OMS. This OMS does not contain references to objects outside of the OMS, thus snapshots are entirely *self-contained*. This self-containedness property implies that all objects, that are needed by the user, must be present in the snapshot. A big advantage is that history recording in **CAMERA** is complete: all objects are automatically version-controlled. In systems where users have to indicate manually which objects must be recorded it frequently happens that certain objects are omitted. For example, tools such as compilers are frequently not version-controlled. This is unfortunate because new versions of these tools may not work with old versions of input data. New tool versions may have different input requirements or contain bugs that are not compatible with a previous version. It is also possible that the versioned data depend on the presence of certain bugs in the tool that may have been removed in a later version.

Snapshots are **immutable**. Once created they cannot be changed. When a snapshot later turns out to be unsatisfactory (e.g. because it still contains bugs), it is not possible to modify it. In such a case a new snapshot must be created, and the old one can be deleted. The immutability of snapshots guarantees a high degree of reproducibility. It is not possible to accidentally modify or delete important versions of certain objects, unless all snapshots in which they occur are deleted.

Every object in the Snapshot OMS has a world-wide unique identifier. This identifier is not changed between snapshots. The same object can be present in multiple snapshots with potentially different values. The object identifier of the object is the same and can be used to determine whether two entities in two different snapshots indeed represent the same value. When a new object is created a new object-identifier is allocated that is globally unique, both in time and space. When two objects in two different snapshots have the same object-identifier, it is therefore guaranteed that there must have been some common ancestor snapshot in which this object was created.

The ancestry of snapshots is recorded by registering the predecessors of each snapshot. This can be used to trace the evolution of the objects inside the snapshots. The snapshot history also gives an overview of the activities, it can easily be determined which objects were modified between two points in time.

**Advantages of the snapshot model**

Snapshots offer the same kind of advantages as composite objects. All versions of objects in the snapshot have actually been used together, and this makes it likely that these versions are mutually consistent.
Snapshots offer version control on all types of objects. Traditional version management systems mainly provide version control on text files. In CAMERA all object types are automatically version controlled, including objects that contain binary information such as compiled programs. Furthermore, version control is also applied to relations and type information, because these are also represented in the OMS. In traditional systems directory information, which can be seen as a primitive form of relations, is not version-controlled. Thus operations such as renaming files and directories are not supported by the version control mechanism. In CAMERA the equivalent operations are also recorded.

Making backups and restoring them is very simple, because every snapshot is self-contained and immutable. All relevant objects are automatically recorded when a snapshot is stored, it is not possible to accidentally forget some objects. With normal backup/restore procedures it is easy to overwrite existing modifications. When an object has been changed since the last backup it could easily be overwritten by subsequent restores. Due to immutability every snapshot has a separate identity, and its contents cannot be changed. This guarantees that it is not possible to overwrite old states.

Because all objects in a snapshot are version-controlled there is no longer a distinction between version-controlled and non-version-controlled objects. This means that version control can be completely transparent for all tools. They can operate in precisely the same fashion as they would in a non-version-controlled OMS.

One of the seemingly obvious problems with integral version management are the storage costs. If a separate copy is made for each snapshot of all objects in the snapshot, this will take a huge amount of storage. Chapter 4 describes some techniques to reduce these storage requirements. Under certain circumstances these techniques can be more efficient than the versioning techniques for composite objects that are used in more traditional systems, because they are better in avoiding unnecessary copies. Many traditional OMSes cannot handle implementations techniques such as delta compression very well. With integral version management it is possible to use these techniques in a much more transparent fashion.

Versions of objects are never developed in isolation. They are always developed within a context of specific versions of other objects. In many version control systems only individual objects are version-controlled. This makes it difficult to determine the relationships between objects because it is not always clear which versions of the objects were used to develop a given version of an object. Without this information it is frequently difficult to understand why and how specific modifications to certain objects were made. Under integral version management this context is automatically recorded.

The contents of a snapshot is not distributed. Either a snapshot is locally available, in which case all of its component objects are available, or it is not available. Furthermore, snapshots are self-contained, they contain all objects that are needed to work with the contents of this snapshot. These properties are attractive in a loosely-coupled distributed environment. Users only need the snapshot with which they want to work and do not have to worry about the availability of individual objects, or other snapshots. These properties
also make it much easier to design tools for use in the OMS, because they need not contain any special exception handling for network failures. Immutability is highly attractive in a loosely-coupled distributed environment, because problems due to modifications to global objects do not exist. This is one of the reasons that some recent distributed operating systems (e.g. COSMOS [NBMW89]) are also based on the notion of immutable objects.

**Disadvantages of integral version management**

The decision to use snapshots as basis for a version management system is a radical one that has its own disadvantages. One of the main disadvantages is that this approach cannot easily be integrated with existing systems. Snapshots are made of a self-contained OMS that cannot contain external references, therefore the resulting system is closed. Furthermore, because versions of an entire OMS are recorded, version control facilities must be incorporated into the implementation of the OMS itself. So it is difficult to build this system on top of existing OMSes. Some efficient implementation techniques are described in chapter 4.

Other disadvantages are caused by the fact that history is primarily recorded on the level of the entire OMS, and not on individual objects as in most other systems. For certain applications this might be cumbersome, because this makes it more difficult to trace the history of individual objects. Also snapshots make it more difficult to combine versions of objects that have never been used together before. These versions must explicitly be extracted from their snapshots and combined together in a new snapshot. **CAMERA** contains several mechanisms for combining information that will be described later on, e.g. pieces, merging, exchange of transformations.

Snapshots are not internally distributed across sites in the network. This implies that it will frequently be necessary to send an entire snapshot over the (possibly slow) network. It is clear that simple optimisations can greatly reduce network traffic, by avoiding the transmission of objects that are already available at the receiver or by sending delta's with respect to such objects.

### 2.4 The Snapshot OMS and environments

**CAMERA** contains an OMS called the **Snapshot OMS**. Instances of the Snapshot OMS act as the users' workspace. Versions of the Snapshot OMS are recorded as snapshots. The Snapshot OMS is self-contained because it contains no references to objects outside the Snapshot OMS. Thus different Snapshot OMSes are entirely disjoint.

Instances of the Snapshot OMS are accessed through **Environments**. An environment represents a private workspace for one or more users. An environment determines the current state of a Snapshot OMS, the contents of which can be stored as a snapshot. The environment offers a view on a Snapshot OMS. All users that use the same environment see the same view. The modifications that are made through this environment are immediately
visible to all its users. Environments have a persistent state, the current state is saved between sessions in the form of snapshots.

Environments provide completely private workspaces. Different environments are completely disjoint, modifications within one environment are completely hidden from all other environments. Traditionally, data is normally shared between all users, and separate mechanisms are needed to obtain isolation. In CAMERA the situation is reversed, isolation is default, and mechanisms are needed for sharing.

The modifications by the user within an environment will lead to the creation of a new snapshot that can be seen as a successor of the previous snapshot of this environment. The ancestry of snapshots is recorded in the successor relation. Every snapshot has a timestamp, a label that uniquely identifies this snapshot. The successor relation determines a partial ordering on time-stamps. Branches in the successor relation can occur due to parallel developments when multiple groups have used the same snapshot as basis for their developments. Merging different lines of development can lead to converging branches where multiple snapshots have a common successor.

CAMERA does not only record the successor relation between snapshots, it also records the OMS-operations that were performed as transformations. A transformation logs all operations that change the contents of the Snapshot OMS. Thus history is recorded in two different ways in CAMERA: snapshots are used to record state, while transformations are used to record operations. Transformations can be replayed on an arbitrary snapshot, in other words they act as functions that take a snapshot as input and return a new snapshot that is the result of replaying its operations on this snapshot. Transformations can also be created manually. Users can take existing transformations and edit these to obtain new transformations.

Arbitrary snapshots can be used as the current state of an environment (see figure 2.1). This gives CAMERA time-machine like capabilities, because the entire state of the environment can be restored to every historical state that has been recorded as a snapshot. This property also supports a very powerful undo operation. The user can simply undo all modifications to all objects by selecting the snapshot that was made just before these operations. This feature is very useful for correcting mistakes and to provide support for explorative programming where multiple alternative approaches are prototyped and compared to determine which approach is superior.

The undo operation is made even more powerful due to the existence of transformations. Frequently the operations that must be undone are not the most recent ones, but a different subset of the operations that were performed since the previous snapshot. Simply using an old snapshot will undo all operations, even the beneficiary ones. Transformations record all operations that have been performed. The beneficiary operations from this transformation can be selected and be performed on the original snapshot. Thus snapshots and transformations together provide a powerful undo/redo mechanism.
Figure 2.1: The time-machine
Different states of the Snapshot OMS are stored as snapshots. Users can “go back in time” by using an old snapshot as the current state of an environment.

Variant management

Many version management systems do not make a clear distinction between the notion of variants and the notion of historical versions. Several authors (e.g. [MSK89, Rei89]) have argued that these notions should be treated as orthogonal to each other, and we have followed this approach in CAMERA. Historical versions are recorded in snapshots, while variant management must be modelled in the OMS itself.

Currently CAMERA does not provide a built-in variant mechanism. The OMS does provide important building blocks e.g. attributed objects, relations and support for operations. Therefore, it is conceptually simple to adapt existing systems, such as Adele [BE86, EGK84] or DSEE’s System Modeller [DSE86, LC84], to the CAMERA OMS. A similar approach was taken with VMCM [BLM91] an implementation of Adele on top of PCTE.

Variants are frequently used for two different purposes: to handle different hardware or software configurations and to support different lines of development, e.g. multiple releases that are maintained simultaneously. Variants for multiple hardware/software are best modelled in the OMS. However, different lines of development are probably better modelled in CAMERA by creating a different environment for each line of development. Thus for each release of a product a separate environment would be created. Maintenance for this release would be performed within this environment. In traditional variant management systems all versions of objects for a specific release must somehow be specified,
e.g. by maintaining a list of which versions are current for a specific release. When an object in a specific release is updated this list must be updated so that the new version becomes current for this release. This is sometimes cumbersome, it is not easy to determine to which release a specific version belongs, and it is not always easy to determine which versions currently belong to the release. When each release has its own environment these problems do not arise since there is precisely one version of each object in the environment. Exchanges between different lines of development, e.g. bug-fixes that must be applied to multiple releases are handled by the standard CAMERA mechanisms for exchanges among environments, such as piece transplants and merging.

2.5 The Album OMS

There are certain kinds of data in the CAMERA system that cannot be recorded inside snapshots. The main problems are mutability and references among multiple snapshots (Remember that snapshots are immutable and self-contained). We will give a few examples. The status of a snapshot, e.g. untested/tested/released is mutable, it can change during its lifetime. Also it must be possible to reverse status changes if errors in the contents of the snapshot are detected at later points in time. It is also difficult to record in a snapshot the set of snapshots that are currently available on a given site because this set is both mutable and contains references to multiple snapshots. Similar problems occur with recording the current snapshot of an environment, and the successor/predecessor relation among snapshots.

It would have been possible to add some special purpose features to CAMERA to store this type of information, but we decided that it would be better to add a general purpose mechanism to represent such information. Therefore, we added a second OMS to CAMERA. Because the main function of this OMS was to collect (information about) snapshots it was named the Album OMS.

The Album OMS is a general purpose OMS in itself, and can therefore be programmed to record and manipulate information about snapshots in a very general way. In the Album OMS, snapshots, transformations and environments are represented as objects. Thus the deletion of a snapshot can be accomplished by deleting the corresponding Album object. The successor relation between snapshots is also recorded in the Album OMS. Album OMSes are not internally distributed, they are local to a given network site (see figure 2.2). Album OMSes on different sites can communicate with one another.

Summarising, CAMERA provides an object management system at two levels. The first OMS is the Snapshot OMS, states of which are stored as snapshots in the system. The Snapshot OMS provides a “relational” object base. The second OMS in CAMERA is the Album OMS (see figure 2.3). In this OMS individual snapshots are represented as objects. The Album OMS is used to store information about developments and is not subject to version control.

The basic structure of the Album OMS is very similar to that of the snapshot OMS. Both OMSes will be described in chapter 3.
2.6 Communication and distribution

In some aspects CAMERA is very different from many existing systems. One of these points is the extent to which data is shared in the system. In most computer systems by default data is shared among users, they will use common parts of a shared file system to do their work. In CAMERA this is more difficult because snapshots are disjoint, and the snapshot OMSes as presented by an environment are thus also disjoint. One important reason for the design decision to base CAMERA on the concept of completely disjoint workspaces is that in a loosely-coupled network this frequently represents the physical reality. When the network connection is broken it is not possible to communicate with other sites, and workspaces are indeed physically disjoint.

The contents of a snapshot is not partitioned across the network, either an entire snapshot is available locally at a given network site or it is unavailable. Thus there is no such thing as a partially available snapshot, due to network disconnections. This greatly simplifies
Figure 2.3: The Album OMS
The Album OMS contains objects that represent snapshots, environments etc., and relations between the Album-objects.

certain matters for the users, since they need not be concerned with the location of parts of the snapshot. Because snapshots are self-contained they contain all objects that are needed to work with (parts of) the snapshot, users cannot accidentally forget to copy objects to their local site. This is desirable since the network might not be available when they detect that certain objects are missing. A further advantage of this self-containedness is the simplification of the operations on the OMS. If certain objects can be unreachable due to network partitioning, all OMS-operations must be able to handle these exceptions. But because snapshots are self-contained such exceptions cannot occur in CAMERA.

For similar reasons the Album OMS is also not distributed. Different nodes will have different instances of the Album OMS, that can communicate with one another.

Because data are not automatically shared among users in CAMERA special mechanisms for communication must be supplied. There are several ways in which users can use the
results of each other's developments.

The first way closely resembles the traditional way of working, in that users completely share the same workspace. This can be accomplished by letting them use the same environment. The modifications of each user are completely visible to all other users in this environment.

Developments can also be shared by using existing snapshots. Users can select snapshots that are available in their local Album OMS and use it as a basis for further development. If a snapshot is not locally available it can be copied from a remote Album OMS.

It is also possible to merge two existing snapshots (see chapter 7) to obtain a new one that contains the desired parts of both snapshots.

A different form of communication is based on the use of transformations. Transformations can contain a set of operations that have been performed in a certain line of development. By replaying the transformations on a different snapshot, the results of the line of development can be incorporated into this snapshot. Existing transformations can be edited, and it is also possible to create a transformation entirely from scratch. Transformations are stored as objects in the Album OMS, and can thus be copied to a remote Album OMS. Normally, a snapshot will be made of an environment before a transformation is applied to it. This makes it very easy to undo the transformation, if it turns out to have undesirable effects.

CAMERA also provides a specialised communication mechanism in the form of piece transplants (see chapter 6). A piece is a special type of composite object that has been designed for the communication of a group of related objects between environments.

2.7 Working with CAMERA

We will now illustrate the way CAMERA can be used with a small example where a small group of programmers is working on small program system that consists of several programs together with documentation.

2.7.1 The OMS

In the Snapshot OMS different classes will be created to hold the different kinds of objects that are needed. For example, there is a class CSource that holds sources that are written in C, similarly there is a class TeXSource that contains the documentation that is written in \TeX. Both of these classes are sub-classes of the class Edit that is the class of all editable text files. Other classes are created for other types of objects, e.g. for pictures. Relations are used to record the relationships among objects. There is a relation documents between a program and its documentation, that can be used to find the documentation for a program and vice versa. Relations are used to record the modules which form part of a program, and the pictures that must be included in the documentation. Methods are used to describe the operations on the object. Both TeXSource and CSource define an operation compile that respectively invokes \TeX and the C compiler. The method edit is also defined on
both. This operation is inherited from the class `Edit`. When a syntax directed editor for C files is available the method `edit` could be defined in class `CSource` to invoke this editor.

The built-in derivation management guarantees that derived values such as the compiled program and the formatted documentation are always up-to-date with all of their inputs. Normally the user will not invoke the compiler explicitly, but will simply attempt to run the program. The derivation manager will then automatically recompile the program when necessary. Derivation management also works for multi-stage derivations, e.g. when the program is used on a test dataset, the test output will also be re-computed when the source of the program is changed. Changes in the structure of derivations, e.g. the addition of a new module to a program will be automatically detected by the derivation manager.

2.7.2 Using environments

Normally each user will have a private environment. These environments can either all be located on the same Album OMS or they can be divided among multiple Album OMSes that are located on different loosely-coupled computers. Modifications are made within their environment and sent to the others when they are deemed sufficiently stable. Communication between environments can take several different forms. One exchange mechanism is the exchange of snapshots. A user selects an arbitrary snapshot and uses that snapshot as current snapshot for her environment. If the snapshot is not available in the local Album OMS it must first be copied from another one. This mechanism is used when a developer wants to continue work on a different computer. Another exchange mechanism is the piece mechanism. Objects are grouped in pieces for exchanges. For example, the documentation for one program is treated as a piece. The documentation consists of multiple objects, e.g. to represent different chapters, indices, pictures. All these objects are grouped into one piece and the entire piece is then exchanged in a single operation. A third exchange mechanism uses transformations to exchange developments. The transformation that contains the modifications of one user is sent to others and then replayed on their environment. The last exchange mechanism uses merging. Here two transformations that represent two different lines of development are merged to obtain a new transformation that represents the union of both developments. The merge procedure will detect conflicting modifications and will help resolving them. When mistakes are made during merging these can easily be detected and corrected by comparing the snapshots of before and after the merge.

Apart from the environments for individual developers, a separate “base-line” environment is created that holds the current approved base-line of the entire product. This environment holds the version of the product that is used as basis for development activities. Before developers start with a modification they will first copy the objects from the base-line environment to their personal environment. The base-line environment is updated with the modifications from the individual participants after they have been tested. In larger teams there will be a librarian who is responsible for these updates. New versions are only added to the base-line environment when they have been approved by the librarian.
In smaller teams there will be no need for a special librarian and all programmers could be able to modify the base-line environment. This is much less dangerous in CAMERA than in most other systems, since the effects of erroneous modifications to the base-line can be easily undone by reverting the environment to the snapshot previous to these modifications. When several releases of the product are being maintained, there will in effect be multiple base-lines, one for each release. This can be handled by creating a separate environment for each base-line. Carry-overs between base-lines (modifications that must be made to multiple base-lines) are handled by the standard mechanisms for exchanges between environments. For larger projects that are organised in multiple sub-projects it may be advantageous to have such base-line environments for each subproject, plus a base-line for the entire project. This would give a similar structure as the parent and child environment in NSE.

Apart from the base-line environments it may be useful to have special tool environments that contain the most recent versions of important objects that are not part of the product itself such as tools etc. When a new version of a tool comes around it will be put in the tool environment and users can copy the tool from this environment. Each tool will be represented as a piece, so that users can obtain all objects and relations that are needed for this tool by copying this piece. Multiple pieces that contain tools can be grouped into larger pieces, for example a large piece could be created to hold all \TeX{} related tools such as bibtex, dvi drivers, Metafont and the \TeX{} program itself. The tool environment could also be used as a repository for class definitions and relations.

2.7.3 Development policies

CAMERA is well suited for development styles based on the copy/modify/merge paradigm. Developers copy the objects they want to modify from a central repository environment to their private environment. They can then modify these objects and finally merge them back into the repository.

A frequently used development policy uses locking to coordinate activities. A user must first lock an object before making any modifications. A locking policy guarantees that there is at most one person with modification permissions for any object. In our view locking is a policy, that users may choose to use or not. Locking can be a sensible policy for larger teams on a tightly-coupled network. However, locking is probably not a very good policy in a loosely coupled environment because access to the central repository is needed for check-in/check-out. Therefore, CAMERA does not have a built-in locking mechanism, but it does supply the mechanisms to support locking as a policy. This could be implemented as follows. A special repository environment is created that holds all the objects, and records locking information. Users can check-out objects from the repository, they are then copied to their private environment. The fact that this object is locked plus information about the locker is recorded in (the current snapshot of) the repository environment. During check-in of the modified objects, the system checks whether the user has indeed reserved the objects that are checked in. This locking mechanism can be implemented very easily, e.g. using relations to record the locking status of objects, plus
a small set of methods that copy objects between environments and handle the locking
information.

In CAMERA users are completely autonomous within their environments. The entire
contents of an environment can be modified, and this environment is isolated from the
activities in other environments. This approach is well suited for a loosely-coupled net-
work, because it is not acceptable that users must wait for permissions from a central
authority that may not currently be available due to network disconnections. For exam-
ple, a user may detect a bug in a module that she was not expected to modify but that
urgently needs fixing. This module can then always be modified in this environment, so
that further work can continue. However, the user may not be able to copy this modified
module to a common environment because she was not expected to modify it. This is
the standard method in CAMERA to enforce certain development policies, users are not
restricted in the activities within their own environment but restrictions can be put on the
communications with other environments.

Because the CAMERA system is highly programmable, exchange procedures can be
automated. For example, it is possible to send a specific piece automatically to other
environments, and to have it incorporated automatically in these environments. Triggers
could be used to determine the conditions under which the piece must be transmitted. The
piece will then be transmitted fully automatically when these conditions are met without
further user intervention. For example, a piece could be transmitted automatically to the
central repository when it has been approved by the developer. This mechanism could
also be used to model tight cooperation on a set of objects between two environments.
Here every modification to these objects in one environment is automatically transmitted
to the other one. The transmission procedures should check for conflicting updates, where
an object has been modified in both environments between two exchanges. Similarly,
when a new version of a tool becomes available, it could be sent automatically to all user
environments and incorporated automatically.

The automated incorporation of objects in a receiving environment is only possible when
this environment explicitly permits it. A cautious user might want to examine the new
modifications before incorporating them. In this case the environment will be programmed
to put all modifications in a mailbox for this environment. A less paranoid user may choose
to incorporate all modifications automatically, but wants to be notified by electronic mail
when this occurs. This is not as dangerous as it may look because it is always possible to
undo the harmful effects by reverting to a previous snapshot.
3 The CAMERA OMS

3.1 Introduction

CAMERA contains two different types of Object Management Systems, the Snapshot OMS and the Album OMS. The main purpose of the Snapshot OMS is to store user data, while the Album OMS is mainly concerned with the storage of information about the development process itself. Both OMSes have the same structure and differ mainly in their pre-defined types and operations.\(^1\) This chapter starts with a description of their common features. Section 3.4 will focus on the features that are specific for the Album OMS.

The CAMERA OMS is a true object-oriented data-base: not only does it provide objects and classes, but it also provides mechanisms to describe operations on the objects.

There is a growing consensus (see e.g. [Bro91, KL89, ZM90, SW87, CAC91]) that object-oriented data-bases provide a good platform for many types of engineering applications. Using an object-oriented approach has many advantages. In most engineering environments several types of entities can be distinguished, e.g. Pascal source files, compiled binaries, human-readable text files etc. The operations that can be meaningfully performed on an entity depend on its type. It is normally useless to call the Pascal compiler on a C source file and vice versa.

This is easily modelled in the object-oriented paradigm. Different types of entities are represented by different classes and each class has a set of operations associated with it. The operations can be specialised for the different classes, e.g. the compile operation on a Pascal source would invoke the Pascal compiler, while the compile operation on a C source would invoke the C compiler.

Certain types can be seen as specialisations of others. A C program text could be handled as a specialised kind of human-readable text, and the text-editor can also be used to edit C programs. Object-oriented systems support this relation between types by inheritance (or by delegation). This makes it easy to re-use or re-define operations from the general class in the specialised class.

A large number of different object-oriented data models exists, (see e.g. [KL89, BFM87]). Although most models have similar features there is no consensus about which models

\(^1\)The term OMS will be used in the rest of this chapter when describing features that are common to both OMSes.
would be most appropriate for specific applications. Consequently we have defined a
model which has enough characteristics of modern systems without being too complex.
There were several reasons why we could not simply take an existing OMS implementation
and use that as a basis for \textit{CAMERA}. The first reason was the implementation of snapshots.
It is obvious that snapshots cannot be implemented in an efficient way unless snapshots
share storage of common values. This can only be accomplished in existing OMSes if we
can have access to the implementation of the object storage. Another reason is derivation
management (see chapter 5). Derivation management as described will only work if
\textit{CAMERA} can intercept all operations on the OMS. Similar facilities are also needed for
transformation logging.
We decided not to use the data model of an existing OMS, but instead decided to design
a simple OMS specifically for \textit{CAMERA}. Existing OMSes had some drawbacks for our
purposes. They either lack facilities that we needed for \textit{CAMERA}, such as the ability
to define new operations (e.g. PCTE). Other OMSes contain elaborate features such as
extensive query languages, that are not really useful for \textit{CAMERA}, or that attempt to
solve problems that were already solved differently by \textit{CAMERA}, such as distribution
facilities and support for composite objects.

3.1.1 The data model of the OMS
Similar to most OMSes, we make a distinction between objects and (attribute) values.
Attribute values are primitive entities such as strings and numbers. Attribute values them-

selves are immutable and do not have a separate identity. For example, the integer “42” is
an an attribute value. This number has no separate identity; all “42”’s represent the same
value, that cannot be modified.

Objects, on the other hand are not values because they can be modified, and do have a
separate identity. Objects capture an encapsulated state. Changing the state of the object
will not change its identity. Thus if we have an object that has an attribute with value “42”,
we can change the state of the object by setting the attribute to a different value, e.g. “63”. The
identity of the object is not changed by this operation, it is still the same object but
with a different state. Different objects with the same state can be distinguished because they
have different identities.

All objects are uniquely identified by an object identifier. Object-identifiers are globally
unique. When a new object is created its object-identifier is guaranteed to be different
from all existing \textit{CAMERA} object-identifiers in the entire world.

Every object has a number of attributes. Every attribute has a name and a value. Attribute
values are typed and \textit{CAMERA} provides a number of pre-defined attribute types such as
strings and numbers (see section 3.1.3).

All objects belong to a class. The class of an object determines the internal structure of the
object: the number, names and types of its attributes, and the operations, called methods,
that can be on the object. The model supports encapsulation. Methods associated with
other objects cannot access the attributes of an object immediately; this can only be done
via the methods of this object. Similar to most object-oriented systems, classes themselves are also represented as objects.

The CAMERA OMS is intended to be used as a coarse grained OMS. Small entities such as numbers will not normally be represented as individual objects, as is the case in fine-grained object-oriented systems such as Smalltalk. We expect that objects in the OMS are comparable in size to files in traditional file-systems. One reason to use coarse grained objects was that it is difficult to implement a persistent fine-grained OMS. Another reason is that the number of objects in a fine-grained OMS rapidly grows, which makes it difficult to comprehend and manipulate its contents.

Relations
Some mechanism to record references is needed because objects never exist in isolation from one another. Indeed all OMSes offer some mechanism to record these references explicitly.

The CAMERA OMS provides relations (i.e. two-way references) as a basic mechanism. This contrasts with some other object-oriented systems (e.g. Gemstone [BMO+89], Orion [BCG+87]) that are based on one-way references (links). We believe that it is important to have relations as a fundamental mechanism in CAMERA for the reasons explained below. In this respect CAMERA is similar to several other DBMSes, e.g. Iris [F+88], PDM [MD86], Cactis [HK88], PCTE [PCT], and Postgres [Se87], that also provide relations as a fundamental mechanism.

From a theoretical point of view, links (one-way references) appear to be sufficient because relationships can be implemented as pairs of links. However, in a very large number of cases it is highly useful to use these links in both directions. We found that in most of our examples, there were legitimate cases where it was necessary to find all objects that referred to a given object. For example, for a specific program it is important to know its components, but for a given component we would also want to know to which programs it belongs. Similarly, for a program we would like to be able to find its documentation and vice versa. Theoretically, it could be determined which objects contain a reference to a given object by iterating over all objects and examining their references. But this approach is both cumbersome and slow.

Another disadvantage of having only one-way references has to do with the implementation of relationships (two-way references) in such a system. Relationships must be implemented with pairs of links. This can lead to problems when objects are exchanged among users. For each relationship the invariant must hold that for each link in the relationship, there is always a reverse link. For an example that illustrates the problems with this approach during exchange of objects see figure 3.1. This picture describes the effect of transmitting an object B from one workspace to an other one. The state of the receiving workspace before and after the transmission is shown. In the sending workspace a relationship has been established between objects A and B, while in the receiving workspace initially a relationship between A and C exists. When object B is transmitted, it has a
Figure 3.1: Problems with relations as pair of links when exchanging objects
This picture shows a potential problem with using pairs of one-way links instead of relations. Object B is sent from one workspace and incorporated in the receiving workspace. The state of the receiving workspace before and after transmission is shown. Afterwards there is a link from B to A but not the other way round.

link to A, but there is no backward link. Thus if objects are transmitted in this way the relationship invariant will be violated. The solutions to this problem will have to treat a pair of links as a single unit, which is equivalent to adding relationships as a basic mechanism.

Some OMSes (e.g. PCTE) only represent individual tuples from a relation, in the form of one-way or two-way references, but not the relation, i.e. the entire set of tuples, as a whole. In CAMERA relations themselves are also represented in the OMS, in such a way that the contents of the entire relation can be queried. This makes it easier to define global queries on the contents of the OMS. In systems that do not permit global queries it is necessary to navigate explicitly through the OMS in order to find specific information. For example, consider the query to find all objects with a given name in all directories. In many systems this query can only be expressed by navigating through the entire directory graph. This can be expensive and difficult to write, especially when directory structures do not form a tree. If the directory information is specified as a relation, this query can be expressed more easily, and can probably be executed more efficiently as well.

The OMS differs from some object-oriented DBMSes in that attributes cannot refer to other objects. References among objects are always made by using relations. This is appropriate since we expect that two-way references will be used by most of the applications of
CAMERA. The need for a separate mechanism for one-way references has not appeared so far.

3.1.2 Class definitions

The internal structure of an object (i.e. its attributes) is described by the template of its class, plus the templates of its super-classes. A template consists of zero or more attribute templates. Every attribute template consists of the name of the attribute, the type of this attribute, and an optional initial value, that will be assigned to the attribute when a new instance is created. A list of the possible attribute types is given in section 3.1.3.

Every class has a name, a label that uniquely identifies this class (see section 3.3.4).

The OMS provides multiple inheritance between object-classes. Every class is related to its immediate super-classes by the super-class relation (see section 3.1.4 for a description of relations):

**System-Relation** super-class: Class $\leftrightarrow$ Class

The classes are partially ordered by the super-class relation. The full set of all super-classes of a class can be obtained by taking a transitive closure of the super-class relation. It is forbidden to make a class an (indirect) super-class of itself.

A sub-class inherits all the attributes and methods from all its super-classes. Attributes cannot be re-defined in a sub-class. Similarly, it is illegal to inherit conflicting attribute definitions from two distinct super-classes by using multiple inheritance. For each attribute in a given class there must be a unique class (either the class itself or one of its super-classes) in which this attribute is defined.

Class definitions are presented in the following way:

```
Class Edit Superclasses Object{
   "Class for editable objects"
   last-modified: integer
   value: string := ""
}
```

This defines the class `Edit`, that is a sub-class of `Object`. It has two attributes: `last-modified` of attribute type `integer` and `value` of type `string` with as initial value the empty string.

Class definitions have an optional comment (a string) that can be used describe the purpose of the class. The root of the object-class hierarchy is the class `Object`. Thus every other class is ultimately a sub-class of `Object`.

In general, in most places where an instance of a given class is needed, instances of its sub-classes can be used as well. For example, this can be used in relations. Thus a relation:

```
Relation ObRel: Object $\leftrightarrow$ Object
```

can be used to relate objects of all classes.

Classes are represented as objects in the OMS, and are instances of the class `Class`. The class `Class` is an instance of itself. Every class object has an attribute template that contains the class-template for instances of that class.
Class Class Superclasses Object {
    "Class for classes."
    template: pair := ()
}

There is a special relation that relates each object with its corresponding class object:
System-Relation instance-of: Object ↔ Class

3.1.3 Types

There are several places where a specification of a type is used in the OMS. There are two different kinds of types: attribute types and object-classes. Types are used in the specification of attributes for a class, the arguments and results of methods, and in the specification of relations. As a convention, attribute types have a name that is entirely lowercase (e.g. string), while object-classes are denoted by capitalised names (e.g. Object).

The following attribute types are available in the current prototype:

<table>
<thead>
<tr>
<th>attribute</th>
<th>Union of all number types</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Integer number</td>
</tr>
<tr>
<td>integer</td>
<td>Real number</td>
</tr>
<tr>
<td>real</td>
<td>Rational number</td>
</tr>
<tr>
<td>rational</td>
<td>Complex number</td>
</tr>
<tr>
<td>complex</td>
<td>Single character</td>
</tr>
<tr>
<td>char</td>
<td>String of characters</td>
</tr>
<tr>
<td>string</td>
<td>Scheme pair</td>
</tr>
<tr>
<td>pair</td>
<td>Union of pair and null</td>
</tr>
<tr>
<td>list</td>
<td>Scheme vector</td>
</tr>
<tr>
<td>vector</td>
<td>Scheme symbol</td>
</tr>
<tr>
<td>symbol</td>
<td>Relation value</td>
</tr>
<tr>
<td>relation</td>
<td>Time-stamp of snapshot</td>
</tr>
<tr>
<td>timestamp</td>
<td>Long string</td>
</tr>
<tr>
<td>bstring</td>
<td>Union of Scheme symbol and string</td>
</tr>
<tr>
<td>symbol-or-string</td>
<td>Scheme procedure</td>
</tr>
<tr>
<td>scheme-procedure</td>
<td></td>
</tr>
</tbody>
</table>

Apart from the attribute types it is also possible to specify object-classes. The name of the class is used to specify the type of an instance of this class, or one of its sub-classes. Thus all objects can be said to have type Object, because Object is the super-class of all classes.

The special type * can be used to denote any (single) type. It can thus be used to designate both attribute types and object-classes. Finally, the special type ** can be used to designate any list of *. It is used in the specification of heterogeneous relations and optional arguments.
3.1.4 Relations

Relations are sets of n-tuples. An individual n-tuple is called a relationship. Every relation determines what n-tuples are allowed as member of the set. Individual tuples are not modelled as objects, they are simply treated as values.

There are two kinds of relations: homogeneous relations where all n-tuples must have the same length and structure (i.e. all corresponding columns in the tuples have same type), and non-homogeneous relations where not all tuples need to have the same length or structure.

A relation is described by a relation-template, a list of types, that determines what relationships are allowed as part of this relation. Relations are represented as objects in the OMS. The operations on relations are described in section 3.3.3.

An example of a (homogeneous) relation description is:

```
Relation directory: Object ↔ symbol ↔ Object
```

This description consists of two parts, the name of this relation (in this case directory), and a relation-template that describes the types of the relationships that can form part of this relation. Every relation has a name, a label that uniquely identifies this relation (see section 3.3.4). The relation-template "Object ↔ symbol ↔ Object" describes a ternary relation, in which each relationship consists of an instance of class Object followed by a symbol (a primitive attribute type) followed by an instance of Object. This example relation can thus be used to mimic the behaviour of traditional file-systems, where symbol represents the name of the second object with respect to the first object that acts as directory.

This template describes the types of the relationships that can form part of this relation. As can be seen from this example the elements in a relation-template can be object-classes (Object) and attribute types (symbol). It is possible to use instances of the sub-classes of a given class, e.g. in this example it is possible to have a relationship: (Dir1, ..., Dir2) between two instances of class Edit denoted by Dir1 and Dir2, because Edit is a sub-class of Object.

Heterogeneous relations have a template in which the last element is the "pseudo-type" **. ** matches an arbitrary number of elements of arbitrary types. Thus, for example, the relation-template "string ↔ **" describes a heterogeneous relation in which the only constraint is that the first element of each relationship in this relation is a string, consequently the relationships can have any size ≥ 1.

There are three different pre-defined classes that hold relations. The first is:

```
Class System-Relation Superclasses Object{
  "Class for system created/maintained relations."
  value: relation
```
} This class is used for relations that should not be modified by the user but only by CAMERA. The value attribute is a value of type relation that is a pre-defined primitive type that contains a set of relationships plus an index. The attribute type relation should not be confused with the class Relation that is described below. Instances of System-Relation are described as follows:

    System-Relation instance-of: Object \rightarrow Class

There is a second class for the relations that can be updated by the user.

    Class Relation Superclasses System-Relation{
        "Class for user created/maintained relations."
    }

The notation for this class has already been described.

Computed relations

The third class for relations is the class Computed-Relation. A computed relation is a relation the value of which is (partially) computed by a number of computation recipes. The value of a relation is the union of the results of these individual computation recipes plus the stored relationships for this relation.

    Class Computed-Relation Superclasses Relation{
        "Class for combination of relation and computed tuples."
    }

A computation recipe is simply a function. The function has one argument, that is the object-identifier of the relation that must be computed. It returns a list of relationships for this relation.

A computed-relation object is related to its recipes by:

    System-Relation recipe-of: Function \rightarrow Computed-Relation

3.1.5 Methods

The OMS contains functions and procedures (collectively known as methods) that can be used to examine and manipulate the internal state of objects.

The attributes of an object can only be accessed by its methods. CAMERA does not provide a programming language, so the implementation of a method is in an external language. The CAMERA OMS only knows the interface of a method. Section 3.2 lists the subroutines that can be used by these external methods to access the contents of the OMS: e.g. routines to read/write attributes and to call other methods.

Methods are represented as objects in the CAMERA OMS. There are two different kinds of methods recognised by CAMERA: functional methods that do not change the contents of the OMS, and procedural methods, that can change the contents of the OMS. Functional methods are used for the specification of derived values (see chapter 5). Functional methods cannot call procedural methods.
Introduction

A method is associated with an object-class and can be used on objects of that class or its sub-classes. Every method can have 0 or more parameters, and 0 or more results. All attribute types and object-classes are allowed as parameter and result types.

(Procedural) methods are notated in the following way:

TeXSource Procedure printDvi (printer:string):
"Print dvi object"

The procedural method printDvi is a method of the object-class TeXSource, it takes one argument printer, and returns nothing.

PascalSource Function compile (options:string): string
"Compile source with Pascal compiler"

The functional method compile takes as input a string containing the compiler options, it compiles the contents of the Pascal source objects, and returns the result of this compilation. This method cannot change the state of the OMS, because it is a function.

Methods are represented by the object-class Method that has two sub-classes Function and Procedure.

Class Method Superclasses Object{
"Class for methods in general."
interface: pair
executable: pair
}

Class Function Superclasses Method{
"Class for methods with functional behaviour."
}

Class Procedure Superclasses Method{
"Class for methods with side-effects"
}

An interface description consists of two lists of types, one for the input-parameters and one for the output values.

Methods can use attribute types and object-identifiers as input parameters and return values, see section 3.1.3.

Within a class all its methods are uniquely identified by their method name (a symbol). This information is stored in the relation:

System-Relation method-of: Class ↦ symbol ↦ Method
"Relates class object with method object. symbol is the selector for this method"

3.1.6 Method invocation

A method invocation consists of three elements: an object called the receiver, a selector and the arguments for this invocation. The OMS determines the appropriate method for this method invocation and then calls the method with the arguments. The procedure for determining the correct method object works as follows.
First the class of the receiver is found, using the instance-of relation. Then this class and all its super-classes are searched to determine whether they have a method that has the given selector as name. This can be done by querying the method-of relation that records the methods for each class. As we have seen the super-class relation forms a partial ordering of the classes, thus it also determines a partial ordering on these methods. It is determined which methods are lowest with respect to this ordering. If there is only one such method, this is the method that will be used. Otherwise the invocation is illegal. The example class hierarchy shown in figure 3.2 illustrates this process. If we use P or Q as selector, and ObjectA as receiver the corresponding methods from ClassA will be invoked. Similarly, the selectors R and S will select the corresponding methods from ClassB and ClassD respectively. Note that an illegal situation would arise if selector R was sent to ObjectA when there was also a method for R defined in ClassC.
Under certain conditions it may be necessary to invoke a method that is not reachable in this way. For example, it is possible in our example that we want to use the method \texttt{P} in ClassB for the implementation of method \texttt{P} in ClassA. A special primitive \texttt{send-super} is available for this purpose. It is defined as:

\begin{verbatim}
define (send-super selector . args)
\end{verbatim}

\texttt{send-super} can only be used inside the body of a method. First it determines the current message class, that is the class in which the currently executing method was defined. By definition the current message class must be the class of the object that received the original method or one of its super-classes. Then it searches for methods with \texttt{selector} in all super-classes of the current message class (excluding the current message class itself), in the same way as for a normal \texttt{send} operation. This method is then invoked with arguments \texttt{args}.

### 3.1.7 Triggers

For some activities it is very useful to let the system automatically perform certain actions when certain situations arise: send a new version of an object to other environments when it is officially approved, recompile a module when the source has changed, send a mail message when a new snapshot is created, etc. In some cases these "actions" can be described in a functional way, e.g. the automatic recompilation can be modelled by treating the compiled program as a function of its sources. However, in other cases the actions inherently have side-effects and can therefore not be modelled as functions, e.g. the automatic distribution of versions to other environments.

For this purpose CAMERA contains \textit{triggers}. Triggers conceptually consist of two parts: a guard that determines when the trigger must be fired, and an operation that must be executed.

Triggers in CAMERA use the results of functional methods as guards. A guard is specified by an object and the selector of a functional method. The value of the guard is the result of sending this method to this object. The trigger is fired when the value of the guard changes.

All triggers are stored in the relation:

\begin{verbatim}
System-Relation trigger: Object ↔ symbol ↔ Object ↔ symbol
\end{verbatim}

The symbols are message selectors. The first \texttt{Object} and \texttt{symbol} define a functional method without arguments that acts as a guard. When the value of this derived value changes, the corresponding operations are executed. These operations are specified by the second \texttt{(Object, symbol)} pair. The \texttt{symbol} must be a selector for a message that has four arguments: the object on which the guard was defined, the selector of the function that computed the guard, the previous value of the guard expression, and the new value of the guard expression.

A guard is a derived value (see chapter 5). This derived value is evaluated eagerly. Since a dependency set is maintained for all derived values, it will also be maintained for all
guards. This dependency set can be used to determine which triggers could possibly be triggered by a modification to the OMS. It can also be used to determine on which parts of the OMS a trigger depends.

### 3.2 The language interface

This section describes the language interface that can be used to access the OMS. Since the current prototype is written in Scheme [RE], this will be used to describe the language bindings.

The main primitive is the function `send`:

```scheme
define (send object selector . arg-list)
```

This function invokes the method with `selector` on `object` with the arguments in `arg-list`\(^2\). Method invocation is described in section 3.1.6. The code for these methods can use the following functions:

```scheme
define (self)
```

returns the object to which the message was sent.

```scheme
define (msg-class)
```

returns the class in which the currently invoked method was found.

```scheme
define (send-super selector . args)
```

This operation is described in section 3.1.6.

```scheme
define (attribute-get att-name)
```

retrieves the current value of the attribute named `att-name` from `self`.

```scheme
define (attribute-set att-name value)
```

sets the current value of the attribute named `att-name` from `self` to `value`.

There are two functions that can be used to retrieve the objects that represent classes and relations (see also section 3.3.4).

```scheme
define (lookup-class class-name)
```

takes the name of a class (a symbol) as argument and returns the object-identifier of the corresponding class object.

```scheme
define (lookup-relation relation-name)
```

takes the name of a relation (a symbol) as argument and returns the object-identifier of the corresponding relation object.

\(^2\)The scheme notation "define (fun arg . opt-args)" indicates that the function `fun` has a regular (non-optional) argument `arg` and can take an arbitrary number of further optional arguments. During the execution of function calls the actual optional arguments will be evaluated and bound to the corresponding parameter `opt-args`. 
Object
  Class
  Edit
  Method
    Function
    Procedure
  System-Relation
    Relation
      Computed-Relation
    System
    Snapshot *
    Environment *
    Transformation *

Figure 3.3: Hierarchy of the built-in classes
Classes marked with * are only available in the Album OMS.

3.3 Built-in behaviour

The OMS contains a number of standard classes, relations and methods. These are shown in figure 3.3, and will be described in the following sections.

3.3.1 Creating objects

In object-oriented systems there are different approaches for creating new objects. The problem in general is that for some classes parameters are needed when new objects of that class are created. But these parameters cannot be sent in a message to the object itself, because it does not yet exist. So the arguments are sent in a message to the corresponding class object instead. In Smalltalk [GR83] every class has an associated meta-class that is used to define methods that are sent to the class object itself. Thus when the message new is sent to a class object, the new method from the meta-class is used. This approach makes it necessary to define a meta-class for each class. Another approach is to make all classes instances of the class Class and to add the initialisation method to this class. In this approach Class is the meta-class for all classes. This is the approach that we have taken and is similar to the one that is found in XLisp [Bet]. It has as a slight disadvantage that it is not possible to declare more than one creation method for a class, but it is conceptually simpler.

New objects can be created by the method

\[
\text{Class \textbf{Procedure} new (args:**): Object}
\]

"Create instance of class SELF, passing ARGS to invocation of initialize to newly created object."

The procedure new creates a new instance of this class and then sends the message initialize args to this new instance. Classes can declare their own initialize
message. Thus classes can have different behaviour for new by defining their own initialize methods. Because of the way message sending is defined, if a class has not defined its own initialize method, the method of its super-classes is used. The class Object has a default initialize message, that does nothing. Since Object is a super-class for all objects, this is the default method that will be used.

Object Procedure initialize (any:**): Object
    "Default dummy initialize."

3.3.2 Creating classes

New classes can be created by sending the message new to the object Class. Class has its own method initialize.

Class Procedure initialize (name:symbol, template:list, supers:list): Class
    "Initialize class SELF by NAME with instance vars from TEMPLATE inheriting from SUPERs."

The arguments to initialize for a Class is a list consisting of the name of the new class (a symbol), the template for the new class, and a list of its super-classes.

New methods can be added to a class with the message add-procedure and add-function.

Class Procedure add-procedure (name:symbol, interface:pair, executable:pair, info:string):
    Procedure
        "Add to SELF under NAME with INTERFACE the EXECUTABLE as procedure method having INFO(optional) as help. Return new method."

Class Procedure add-function (name:symbol, interface:pair, executable:pair, info:string): Function
    "Add to SELF under NAME with INTERFACE the EXECUTABLE as function method having INFO(optional) as help. Return new method."

3.3.3 Relation-operations

Relations are instances of (sub-classes of) the class System-Relation. Individual relationships are implemented as a Scheme list.

The current value of the relation can be obtained by sending the message value.

System-Relation Function value (): list
    "Return value of relation as list of relationships"

New relationships can be added to a relation by the procedure:

Relation Procedure add (tuples:**): Relation
    "Add to relation SELF TUPLES."

Relationships can be removed from a relation by the procedure:

Relation Procedure delete (patterns:**): Relation
    "Delete from relation SELF tuples matching PATTERNs."
Note that these methods are defined in the class Relation and not in its super-class System-Relation because System-Relations cannot be modified by the user.

A subset of the relationships can be computed with the function:

\textbf{System-Relation Function} \texttt{query (pattern:pair): **
"Query (system) relation SELF by PATTERN."}

A query is a pattern, and \texttt{query} returns a set of all relationships that match this pattern.

A pattern is composed of sub-patterns, that match individual elements of a tuple. A sub-pattern can be:

- a constant: the corresponding element in the tuple must be equal to this constant
- a list of constants: the correspond element in the tuple must be equal to a member of this list
- the special sub-pattern \texttt{*}: matches any value for the corresponding element in the tuple
- the special element \texttt{**}: this sub-pattern must be the last one in the query, it always matches the remaining elements of the tuple, irrespective of length.

A few examples:

\(\{42 \,*\}\) matches all tuples of length 2 whose first element is equal to 42.

\(\{(9, 7)\}\) matches all tuples of length 1 whose first element is equal to 9 or 7.

\(\{17 \,*\,**\}\) matches all tuples of length 2 or longer whose first element is 17.

The function \texttt{query} is very important, since it will often be used as a derived value that acts as fire-wall in derivations (see chapter 5).

3.3.4 Naming objects

Every object has a unique (system generated) object-identifier. This is of course not suitable for the user: objects should have more understandable names. The OMS contains a built-in naming scheme for classes and relations.

The name of a class is stored in

\textbf{System-Relation} \texttt{Class-Name: symbol \leftrightarrow Class}

The name of a relation is stored in

\textbf{System-Relation} \texttt{Relation-Name: symbol \leftrightarrow System-Relation}

Directories

New naming schemes can be added on top of the OMS easily, using relations and objects. One such scheme is provided by the OMS and is similar to the scheme used by Unix. There is a relation:

\texttt{Relation directory: Object \leftrightarrow symbol \leftrightarrow Object}
The first object in this relation acts as a directory. There is no pre-defined "directory" class in the OMS. Any object can act as a directory. The second element in this relation is the name of the object (the third element) in this directory.

The name of an object can be indicated by concatenating the names of its directories separated by slashes e.g. "/usr/bin/cc". There is a special distinguished directory object called root indicated by "/".

There is a primitive routine that converts an object-name to its object-identifier:

```
    define (dir-lookup path-name)
```

The directory relation is used to lookup classes and relations. There are two distinguished objects with names /class and /relation that act like normal directories that contain all classes and relations respectively. Thus the class object for the class Edit can be found under the name /class/Edit. Similarly, the instance-of relation has the name /relation/instance-of.

### 3.4 The Album OMS

The Album OMS contains the objects that are used to describe the history of the development process, i.e. the snapshots. The Album OMS contains some specialised classes and relations that are not present in the Snapshot OMS.

#### Snapshots

Snapshots are represented as objects in the Album OMS.

```
    Class Snapshot Superclasses Object{
        "Class for snapshots."
        timestamp: timestamp
    }
```

There is a pre-defined relation:

```
    System-Relation derived-from: Snapshot ↔ Snapshot
    "The successor relation between snapshots."
```

#### Environments

The working context of a user is represented by an environment:

```
    Class Environment Superclasses Object{
        "Class for environments."
    }
```

Every environment has one current snapshot:

```
    System-Relation env-current-snapshot: Environment ↔ Snapshot
```

Conclusions

Before a user can actually use an environment it must be activated and the identification of the user must be given as argument. The return-value can be used to communicate with the environment process.

\[
\text{Environment Procedure use (user-id:*)}: \text{number} \quad \text{"Use environment SELF under id USER."}
\]

Similarly, there is a message to stop working with an environment.

\[
\text{Environment Procedure stop (user-id:*)}: \text{objectid} \quad \text{"Stop using environment SELF under id USER."}
\]

There is also a method that makes a new snapshot of the current state of the Snapshot OMS for an environment.

\[
\text{Environment Procedure make-snapshot ()}: \text{Snapshot} \quad \text{"Make a new snapshot of environment SELF."}
\]

Transformations

Transformations are represented by objects:

\[
\text{Class Transformation Superclasses Object}{
\text{value: *}}
\]

Transformations can be produced by the creation of a new snapshot. There is one transformation associated with every snapshot and its successor.

\[
\text{System-Relation transformation: Snapshot} \leftrightarrow \text{Snapshot} \leftrightarrow \text{Transformation}
\]

3.5 Conclusions

This chapter has described the facilities of the CAMERA OMS. The OMS is a true object-oriented system. One of the main differences with more traditional repositories such file-systems or object stores such as PCTE is the ability to add new operations to the system. Other facilities are relations, and a class mechanism with multiple inheritance. The Album OMS and the Snapshot OMS have been designed to be very similar.

The CAMERA OMS does not have a very complicated structure. Nevertheless, we believe that it is very powerful, and forms a good repository for the CAMERA system. Initial experiences with the OMS prototype have generally been positive. The prototype will be used for further evaluation of the OMS.
4 Data-structures for integral version management

4.1 Introduction

The previous chapters have explained the notion of integral version management, and how it is realized in CAMERA by recording the state of an entire workspace as a snapshot. This concept is only attractive for version management if snapshots can be implemented efficiently. This chapter describes data-structures that can be used for such an implementation. Because these techniques are rather general, they could also be used to implement integral version management in arbitrary object management systems. They are presented here in an abstract way mostly without reference to the specifics of CAMERA. This section also includes some data-structures that can be used for linear version histories, where each version has only one successor. These data-structures cannot be used in CAMERA, but are included because they are interesting in their own right.

The global structure of the chapter is as follows. Section 4.2 introduces the model that is used to describe the data-structures. A global overview of the data-structures is given in section 4.3. The following sections describe data-structures for linear version history (4.4) and for the non-linear version history (4.5).

4.2 A model for integral version management

In this section gives a model for integral version management which is used in this chapter for the description of the implementation techniques.

The example object model

This section describes the sample object model that will be used in the rest of this chapter. We assume that data is stored in an object management system that is based on a hybrid data model, i.e. a model providing both objects and relations.

Objects are identified by an object-identifier OID. The object identifier of an object remains the same when the object is changed. Every object has a value. The internal structure of the objects is not relevant for the rest of this chapter.

1This section is a modified version of [LF91], that was written together with Gert Florijn.
References are handled via relations. Relations are sets of relationships, n-tuples of object-identifiers or primitive values. Each relation is identified by a relation identifier RELID. In CAMERA the relation identifier is simply the object-identifier of the corresponding relation object, but they are distinguished in this chapter to keep the model more general. We can use relations to describe how version control on inter-object references can be implemented. Relations are also treated separately because references between objects could change at a different rate than the contents of the object. This makes it more efficient to separate the version control of references from the version control of the contents.

Snapshots
Under integral version management versions of the entire contents of an object management system are recorded. Versions of the OMS are called snapshots. Since snapshots are used to record history we will assume that old snapshots (i.e. snapshots for which a successor has been created) are immutable. If errors are discovered in an old snapshot, a new snapshot with the modifications must be created. Possibly, the erroneous snapshot could be deleted.

Every snapshot is uniquely identified by a time-stamp, also known as version identifier. Time-stamps are unique labels, that need not have any relation to actual clock times. Time-stamps have an ordering imposed on them, that describes the successor relation between the versions.

Two different types of history can be distinguished: linear and non-linear version history. In linear version history every version (except for the first and last one) has a single
successor and predecessor. This happens if there is one central version that is developed. The ordering that corresponds with linear history is a total ordering of the time-stamps.

In a non-linear version history, versions can have more than one successor, and more than one predecessor. This version history is characterised by branches and merges. This occurs during parallel development, when several persons use the same version as basis for their work. The ordering in a non-linear version history is a partial ordering of the time-stamps, i.e. the successor relation forms a directed acyclic graph.

As we have seen above inside the OMS every object has a unique object identifier (OID) and likewise every relation is identified by a relation identifier (RELID). Thus a version of an object (relation) can be identified by the combination of its object (relation) identifier and the time-stamp of the snapshot to which this version belongs.

The two most important operations on sets of snapshots, are creation and retrieval. Users can retrieve any existing snapshot, and use that as the contents of their workspace. At certain points in time a new snapshot is created of the entire contents of this workspace. Users can also examine parts of existing snapshots, e.g. to determine the values of objects and relations inside that snapshot.

4.3 The implementation model

An overview of the model for integral version management is given in figure 4.2. Versions of the entire contents of an OMS are stored by the system, and one such version is called a snapshot. Users operate on specific snapshots, each of which contains a particular state of the OMS. The snapshot manager handles operations on these snapshots, and maintains the successor relation between snapshots.

All snapshots are stored in one global snapshot storage, and this chapter describes data-structures for the efficient implementation of this global store. Different techniques can be used for objects and relations, and the following sections will introduce the functionality of these data-structures. This section will introduce a global framework that is used to describe the specific data-structures for linear and non-linear history, in sections 4.4 and 4.5, respectively.

Storing versions of objects

At an abstract level the functionality of an OMS can be described as a data-structure that associates an object-identifier of an object with the value (the contents) of this object, i.e.:

\[
OMS : OID \rightarrow value
\]

If we add integral version management on objects, this can be described as:

\[
Index : OID \times TimeStamp \rightarrow value
\]
i.e. this data-structure maps a combination of an object-identifier and a time-stamp to the value of this object in the snapshot.

A straightforward approach for implementing integral version management is to store a complete copy of the entire snapshot. This is effectively what is done in Smalltalk, where the equivalent of a snapshot is a Smalltalk image. It is obvious that for virtually all practical applications this will soon become prohibitively expensive, unless the number of snapshots remains very small.

A somewhat more advanced solution is based on the observation that successive snapshots tend to be very similar, since most objects and relations will not change between snapshots. It is therefore wasteful to make separate copies of all objects — especially when objects are large — for all snapshots in which the object occurs, since the value of the object will be the same in most of these snapshots.

An obvious implementation technique is to let snapshots share common values of objects. This can be done by adding an extra level of indirection. Every different value of an object is stored in a separate data-structure, and can be identified by a value identifier (VALID). The data-structure now consist of two parts:

\[ \text{VersionIndex} : \text{OID} \times \text{TimeStamp} \rightarrow \text{VALID} \]

and

\[ \text{ValueIndex} : \text{VALID} \rightarrow \text{value} \]
The `ValueIndex` is an index on the values of all objects. In order to retrieve a specific version, first the `VersionIndex` is searched to find the corresponding `VALID`, which is then used as key in `ValueIndex` to retrieve the value of this version. In a functional notation this can be described as

\[ \text{Index}(oid, \text{timestamp}) = \text{ValueIndex} \left( \text{VersionIndex}(oid, \text{timestamp}) \right) \]

The implementation of the function `ValueIndex` will be dependent on the type of the values. With text files, for example, compression techniques such as delta-compression ([Tic85]) can be used. For objects that contain fixed length records the techniques from [KL84] might be used. Reichenberger [Rei91] describes a compression technique that can be used on arbitrary byte strings. A data-structure that can be used to store different versions of a set in a compressed way is described in sections 4.4.2 (linear history) and 4.5.2 (non-linear history).

For objects that are small or that have very dissimilar values, these compression techniques cannot be used: in this case the different values will be stored completely. Very small values (e.g. integers) can even be stored in the `VersionIndex` itself, instead of their `VALID`, thereby avoiding the extra overhead of using the `ValueIndex`.

In the rest of this chapter we ignore the `ValueIndex`, and concentrate on implementations for `VersionIndex`. These are described in sections 4.4.1 and 4.5.1.

**Storing versions of relations**

It is of course possible to treat a relation as an object that has a set (of relationships) as its value, and then to use the techniques from the previous section. Special algorithms would be needed to compress the values of relations. Since relations are sets, we could use the techniques like the ones described in sections 4.4.2 and 4.5.2. However, the speed of access for relations can be greatly improved by defining special index structures, which reflect the common use of the relation.

A standard method for the implementation of relations is as one or more indexes on search keys of the relation. Such an index has the following functionality:

\[ \text{RELID} \times \text{Key} \rightarrow P \text{ Relationship} \]

The index takes a relation identifier plus a search key, and returns the set of relationships with this key. These indexes are commonly found in data-bases but can also be used for the navigational behaviour found in most OMS applications, by using object-identifiers as search keys.

A modified form of these indexes can be used to store old versions of relations. We define a two-level index:

\[ \text{RelIndex} : \text{RELID} \times \text{Key} \rightarrow \text{RVALID} \]
RelVersionIndex : RVALID x TimeStamp → P Relationship

RelVersionIndex implements versions of sets (of relationships). Specialised data-
structures for storing versions of sets are described in sections 4.4.2 (linear history)
and 4.5.2 (non-linear history).
RelIndex can be implemented using traditional indexing techniques, e.g. B-trees [Knu73].
Because these indexes have a key, fast access to the tuples is possible.

4.4 Linear version history

An important special type of history is the linear version history in which each time-stamp
(except for the last one) has precisely one successor. The restricted nature of the linear
version history makes it possible to use more efficient data structures than those for the
general non-linear version history, which will be described in section 4.5.
This section, that describes data-structures for linear version history, is divided in two
parts. The first part (section 4.4.1) concentrates on implementations for VersionIndex, the
data-structure that is used for integral version management on objects, while section 4.4.2
describes a data-structure for storing versions of sets that can be used for integral version
management on relations.

4.4.1 Sparse index

An obvious implementation for VersionIndex is to use a B-tree [Knu73], with the com-
bination of OID and TimeStamp as key. If the index is sorted lexicographically on (OID,
TimeStamp), all entries with the same OID will be adjacent in the index, and ordered by
increasing TimeStamp.
It can be expected that successive entries in the index will tend to have the same value,
since typically an object will not change between two snapshots. In a certain sense these
repeated entries are redundant, and can be omitted. We only store references to values at
those times that objects have changed value.
A more precise definition of when an entry can be omitted is the following. We define that
an entry (oid, t) immediately precedes an entry (oid, t') in the index if there is no entry
(oid, t'') in the index with t < t'' < t'. An entry (oid, t) can be omitted from the index if
there is an entry (oid, t') that would immediately precede (oid, t) if it were present in the
index, such that indexval( (oid, t) ) = indexval( (oid, t') ).
A sparse index is an index where the value of each entry is different from the immediately
preceding entry. In order to handle deletions of objects correctly, the domain of object
values is extended with the special object value null, i.e. an entry ((oid, t) → null) would
indicate that at time t, object-identifier oid was not referring to an existing object.
The insertion algorithm for a new entry ((oid, t) → val) is:
if the index contains an entry \((oid, t') \rightarrow val\) that immediately precedes \((oid, t)\) such that \(\text{indexval}(oid, t') = val\)
then do nothing
else insert \((oid, t) \rightarrow val\) in the index

Obviously, the deletion of an object \(oid\) at time \(t\) is indicated by inserting an entry \((oid, t) \rightarrow \text{null}\) in the index.

The algorithm to retrieve the value of an object \(oid\) at time \(t\) is:

if there is an entry for \((oid, t)\)
then return \(\text{indexval}(oid, t)\)
else if there is an immediately preceding entry \((oid, t')\)
then return \(\text{indexval}(oid, t')\)
else return null

This sparse index can be implemented using traditional data structures, e.g. the ubiquitous B-trees.

**Performance**

The index will only contain an entry for an object, if that object has been changed with respect to the previous snapshot. In general, it is to be expected that most objects will not change between snapshots. Therefore, a sparse index will be much smaller than the corresponding full index that stores all object values for all time-stamps.

Searching the sparse index can be somewhat faster than with the full index due to the smaller size of the index. In a similar way the creation of a new snapshot can be faster because only objects that have been changed since the previous snapshot must be entered into the index.

4.4.2 Versions of sets

This section describes a data-structure that can be used to handle versions of sets for a linear version history, that can be used to store versions of relations. This data-structure is called version-list. A version-list consists of a list of 3-tuples \((el, birth, death)\), where \(el\) is an element of the set, \(birth\) and \(death\) are both time-stamps. The meaning of a 3-tuple is that \(el\) is a member of the set at all times \(t\) such that \(birth \leq t < death\). All elements that are in the set at the most recent time-stamp have a death time-stamp of \(+\infty\).

The 3-tuples in the version-list are sorted on decreasing death time-stamp. This means that all the entries that are currently alive are located near the head of the list. When a new element \(r\) is added to the set at time \(t\), a new entry \((r, t, +\infty)\) is added to the version-list.

When an element \(el\) is deleted at time \(t\) the death time of the corresponding 3-tuple is changed to \(t\). In order to keep the list sorted this 3-tuple may have to be moved to the rear of the list, this can be arranged by swapping this entry with the last 3-tuple that is currently alive.
The algorithm for reconstructing the set of elements for a given time $t$ from a version-list $v$ is:

$$
i := 1;
\text{result} := \emptyset
\textbf{while} \; i \leq \text{size}(v) \land v[i].\text{death} > t \; \textbf{do}
    \textbf{if} \; v[i].\text{birth} \leq t
        \textbf{then} \; \text{result} := \text{result} \cup v[i].\text{el}
        i := i + 1
$$

Observe that we can terminate the search loop when an entry is found for which $v[i].\text{death} < t$ because the list is sorted on decreasing death time. So we know that for all $j > i$, $v[j].\text{death} < t$.

**Performance**

Because old snapshots are immutable it is only possible to insert and delete elements for the most recent snapshot. Therefore, insertion and deletion operations will be $O(s)$ where $s$ is the size of the current version of the set.

From the algorithm that retrieves old versions of the set, it can be seen that newer versions can be retrieved faster than older versions. This is desirable, because it can be expected that recent versions will be used much more frequently than older versions. The time needed to retrieve the most recent version is $O(s)$, where $s$ is the size of the current version of the set, and this is of course the best achievable, if we want to enumerate all members of the set.

### 4.5 Non-linear version history

In the general case, version histories are not linear — i.e. time-stamps can have more than one successor/predecessor — due to branching and merging. **CAMERA** employs a non-linear version history. The techniques that were used in the previous section are not straightforwardly extensible to this case. This section describes some extensions that can be used in the non-linear case as well.

Like the previous section, this section is split into two parts. The first part describes techniques for storing objects. The second part describes data-structures for storing versions of sets that can be used to implement versioned relations, as was explained in section 4.3.

#### 4.5.1 Storing versions of objects

The sparse index technique, that was described in section 4.4.1, cannot be used immediately in the case of non-linear history. The most important problem is that in this case time-stamps are not totally ordered. Therefore, they cannot be used as key in the sparse index. This problem can be solved by adding yet another indirection.
The procedure works as follows. There is one global mapping table:

\[ \text{map} : \text{TimeStamp} \rightarrow \mathbb{N} \]

that is used to convert a TimeStamp to a (totally ordered) natural number. The resulting number is used to index the modified sparse index

\[ \text{VersionIndex}' : \text{OID} \times \mathbb{N} \rightarrow \text{VALID} \]

The total functionality can be described by

\[ \text{Index}(oid, timestamp) = \text{ValueIndex}(\text{VersionIndex}'(oid, \text{map}(timestamp))) \]

The mapping table, map will contain one entry per snapshot. It will thus be relatively small, and could be kept in core. It can be implemented e.g. as a hash table.

The insertion algorithm for the non-linear version history is similar to that for linear version history. However, there is a small difference. If we want to insert an entry in the index for a time \( t \) it is possible that there is another time-stamp \( t' \) with \( \text{map}(t') > \text{map}(t) \). Note that this was not possible in the linear case because the time-stamp that was entered was always the most recent one. A problem arises if there exists a time-stamp \( t'' \) for which there is currently no entry in the index but for \( (oid, t) \) would immediately precede \( (oid, t) \) if it were inserted. In this case we must make a separate entry for \( (oid, t'') \) because it has now a different immediate predecessor. Here is a small example to illustrate this. Suppose that the index initially contains entries for two time-stamps \( \text{map}(t') = 4 \) and that \( \text{map}(t'') = 6 \). If the value of the object \( o \) is the same at both time-stamps, the index need not contain a separate entry for \( (o, 6) \). But if we add a snapshot with time-stamp \( t \) for which \( \text{map}(t) = 5 \) and that has a different value for object \( o \), we must now also add a new entry for \( (o, 6) \), which has of course the same value as that for \( (o, 4) \).

The modified insertion algorithm for a new entry \( (oid, t) \mapsto val \) is:

\[
\begin{align*}
\text{if} & \quad \text{the index contains an entry} (oid, t') \mapsto val' \text{ that immediately precedes} (oid, t) \\
& \quad \text{such that} \ \text{indexval}(oid, t') = val \\
\text{then} & \quad \text{do nothing} \\
\text{else} & \quad \text{insert} (oid, t) \mapsto val \text{ in the index} \\
& \quad \text{let} t'' \text{ be the time-stamp such that} (oid, t) \text{ immediately precedes} (oid, t'') \\
& \quad \text{if} \quad \text{the index does not contain an entry for} (oid, t'') \\
& \quad \text{then} \quad \text{insert} (oid, t'') \mapsto val' \text{ in the index}
\end{align*}
\]

Example

An example sparse index for the data in figure 4.3 is shown in figure 4.4. In this example, we have two objects \( OID_1 \) and \( OID_2 \), and a set of snapshots with time-stamps \( T_a \ldots T_f \).
Figure 4.3: Example data for sparse index. This figure shows a partially ordered set of time-stamps, and the contents of the corresponding snapshots. Each snapshot contains one or two objects. The object on the left has object-identifier $OID_1$ and the one on the right $OID_2$.

The initial snapshot is $T_a$, that only contains the object $OID_1$ with value $W$. In one line of development ($T_b, T_d, T_e$), the object $OID_2$ is created and modified. In the other development line ($T_c$) object $OID_1$ is modified. The final snapshot $T_f$ merges the results of both lines of development.

**Insertion algorithms**

The size of the index $VersionIndex'$ depends on the mapping table in the following way: for all TimeStamps $A$ and $B$ let $d(A, B)$ be the number of objects that are different between snapshot $A$ and snapshot $B$. The TimeStamps in the index are sorted by the increasing value of the mapping function. If we arrange all TimeStamps in a sequence $t_1 \ldots t_n$ such that $map(t_i) < map(t_{i+1})$, the total size of the index is

$$l = \sum_{i=1}^{n-1} d(t_i, t_{i+1})$$

A bad choice for the mapping table will give a very large index. Unfortunately, it is not very easy to discover the optimal mapping table for a given set of snapshots, that gives a minimal value for $l$.

**Lemma:** Finding an optimal mapping function for a sparse index, as described in section 4.5.1 is NP-complete.
\[
\begin{array}{|c|c|}
\hline
T & \text{map}(T) \\
\hline
T_a & 1 \\
T_b & 2 \\
T_c & 6 \\
T_d & 3 \\
T_e & 4 \\
T_f & 5 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{OID} & \text{map}(T) & \text{Object-Version} \\
\hline
OID_1 & 1 & ValW \\
OID_1 & 5 & ValX \\
OID_2 & 2 & ValY \\
OID_2 & 4 & ValZ \\
OID_2 & 6 & \text{null} \\
\hline
\end{array}
\]

Figure 4.4: A sparse index
This table shows an example sparse index and the corresponding mapping function. null indicates a deleted entry. \(ValX, \ldots\) are the value identifiers for \(X, \ldots\)

Proof:
First observe that the corresponding decision problem is in NP, since if we are given an ordering and its corresponding index size, it can be checked in polynomial time whether this ordering has indeed this index size. Since the decision problem is in NP, the problem of finding an optimal ordering is in NP, too.

Now we will reduce a standard NP-complete problem, a version of the Hamiltonian path problem, to our ordering problem. The Hamiltonian path problem can be stated as follows: Given a graph consisting of a set of nodes \(N = \{\nu_1 \cdots \nu_n\}\) and a set of edges \(E = \{e_1 \cdots e_s\}\) without double edges between nodes, find a path that passes exactly once through all nodes.

Now we are going to construct an instance of the optimal ordering problem for an instance of the Hamiltonian path problem. We construct a set of snapshots \(\bar{X} = \{\bar{x}_1 \cdots \bar{x}_n\}\), where

\[
\bar{x}_i = \begin{pmatrix}
z_{i,1} \\
\vdots \\
z_{i,s}
\end{pmatrix}
\]

Every snapshot corresponds with a node in the Hamiltonian path problem. We represent a snapshot by a column vector that contains the values of the objects, thus \(z_{i,j}\) contains the value of object \(j\) at time-stamp \(i\). For this proof we only need objects that can have integer values.

The contents of each \(\bar{x}_i\) is as follows:

\[
z_{i,j} = \begin{cases} 
0 & \text{if } e_j \text{ is connected to } \nu_i \\
i & \text{otherwise}
\end{cases}
\]

This transformation can be performed in polynomial time.

Remember that the distance between two snapshots is equal to the number of objects that have different values in both snapshots. Observe that the distance between two points that
represent two connected nodes is $e - 1$ while the distance for unconnected nodes is $e$. Now there exists a Hamiltonian path if and there exists an ordering of the $X$ such that the total sum of distances between successive snapshots in the index is equal to $(n - 1)(e - 1)$. □

Because the problem of finding an optimal mapping table is NP-complete, for a practical insertion algorithm heuristic methods must be used. Which particular method is most suited depends very much on local branching patterns. The index will be relatively small if every snapshot is similar to its neighbours in the index. It can be expected that every snapshot is similar to its predecessor, thus a simple heuristic would be to insert every snapshot as a neighbour in the index of its predecessor.

It is not always possible to insert a successor of a snapshot with time-stamp $t$ as a neighbour, this happens if $map(t) + 1$ and $map(t) - 1$ are already used in the mapping table. In this case the new snapshot must be inserted elsewhere. A useful (greedy) heuristic, is to insert this snapshot at a place in the index where it causes the smallest increase in the total index size, i.e. between two snapshots to which it is very similar. Simulation experiments with this heuristic (described in [Lip90]) indicate that the size of the produced index is normally very close to the size with the optimal mapping table.

Performance

The performance of retrieval operations on this data-structure can be very similar to that of the standard sparse matrix for linear history. The only extra overhead is one lookup in the mapping table. When the relevant part of the mapping table is kept in core (as it probably will be) the additional overhead is negligible.

4.5.2 Implementations for versioned sets

This section describes implementation techniques that can be used to store versions of sets, in the case of non-linear version history. As was explained in section 4.3 this data-structure can be used to store versions of relations.

Several standard data-structures like lists and hash tables can be used to represent sets. But these data-structures cannot be used to store multiple versions of the set in an efficient way. Two data structures, delta-lists and modified AVL-trees, that are more space efficient will be described in the next sections.

Delta-lists

A first data-structure for the implementation of versioned sets makes use of delta-lists, see figure 4.5. This data structure consists of two parts. Certain versions of the set are stored in full as base versions, using a traditional data structure (hash tables are suitable). Other versions are stored as sets of changes to a base version in the delta-lists. A delta-list is a list of deltas, every delta contains the differences (additions/deletions of elements) with respect to the previous version of the set. Every delta in a delta-list is marked with a
time-stamp. The first delta in every delta-list contains the modification with respect to a fully stored base version.

To retrieve a specific version of the set with a specific time-stamp $T$, first the corresponding base version must be retrieved, and then all delta's that have time-stamps between that of the base version and $T$ must be applied. Thus the average access-time will depend on the size of the delta-lists. If base versions are made more often access-time will get shorter, but storage demands will increase.

The average amount of storage per set with this data-structure is:

$$c_1 \cdot f \cdot n + c_2 \cdot (1-f) \cdot k$$

where

- $f = $ proportion of the versions that are stored as base versions
- $c_1 = $ storage size per element in a base version
- $n = $ average number of elements in a set
- $c_2 = $ storage size per element in a delta
- $k = $ average number of elements in a delta

The average access-time to retrieve a specific version of the set is:

$$\frac{1}{2}(1-f) \cdot d_1 \cdot k + d_2 \cdot n$$
Figure 4.6: Shared sub-tree implementation for sets
The black tree and the shaded tree represent two different versions of the
same set that share common sub-trees.

where \( d_1 \) = time to access one element in a delta
\( d_2 \) = time to access one element in a base version

The procedure for accessing an element, first the corresponding base version is retrieved,
and it must be determined whether the element is part of the base version. This costs \( d_2 \cdot n \).
Then we must apply all deltas, between this base version and the version we want. On the
average there are \( \frac{1}{2}(1 - f) \) deltas that must be processed. Each delta costs \( d_1 \cdot k \).
The proportion of versions that are stored as base versions \( f \) is an important parameter
when implementing this data-structure. It represents a classical space/time tradeoff, a
low value for \( f \) gives a compact data-structure while a high value gives good access
performance.

Trees
Another standard way to implement sets is by using trees, e.g. AVL trees (see e.g. [Knu73]).
This data-structure can be used in a slightly modified form to store different versions of
sets in an efficient way, by sharing common sub-trees. In large trees successive different
versions of a set, that differ in only a few elements, will have large sub-trees in common,
and these can be shared between the different versions, see figure 4.6.
The procedure for inserting/deleting elements is almost the same as that for a standard
AVL tree, but instead of modifying existing nodes in the tree, a new copy of the node
is made that contains the new value. When a node is copied, all of its ancestors must be
copied as well. In each copied ancestor the changed sub-tree will replace the original.
Unmodified sub-trees are shared between the old and the new version of the tree. If, for
example, two successive sets differ only in one element, they will only differ in the nodes
that form the path from the root to that element. All other nodes can be shared between
the two versions.
Every individual tree is a normal balanced AVL-tree, so average access-times are logarithmic in the size of the tree. Also the worst case access-times are logarithmic, since the maximum depth of an AVL-tree of size $N$ is $1.4404 \log_2(N + 2) - 0.328$ [Knu73]. For single deletions/insertions the storage requirement to store the new version is logarithmic in the total size of the set, since all elements of the path to the root must be copied. However, when multiple additions/deletions are made the overhead per individual modification is less than $\log n$.

**Analysis of AVL-trees for versioned sets**

An important question about the implementation of versioned sets with AVL-trees is the amount of storage that will be consumed by this data-structure. This section attempts to give some approximate answers.

The storage requirements are determined by the overlap between successive versions. If they can share many common sub-trees, they will require less storage. So the question is: how many nodes will on the average be shared between a tree and its successors?

The key observation for this analysis is that a node will only be shared with the next tree if the sub-tree that is rooted at this node, is left completely unchanged. Thus, the expected number of nodes that can be shared with the successor of tree $T$ is:

$$\sigma(T) = \sum_{s \in T} p(s)$$

where $p(s)$ is the probability that sub-tree $s$ will not be modified.

In a standard AVL tree, every insertion/deletion will cause the modification of the child pointers of at most two nodes, due to re-balancing. Each node that would be modified in such a way by the standard must be copied for our algorithm. These nodes will be called primary nodes. For each new version of the set we must store its primary nodes, because the corresponding nodes of the old version cannot be modified. But there are also other nodes, called secondary nodes, that must be stored for this new version. Secondary nodes are those nodes that are created because one of their sub-trees was modified. Thus all nodes that are above a primary node are secondary nodes. We assume that the possibility that a node does not undergo a primary change is the same for all nodes and equal to $c$, and that the probabilities for the different nodes are independent from one another. Under these assumptions the expected number of shared nodes becomes

$$\sigma(T) = \sum_{s \in T} c^{|s|}$$

where $|s|$ denotes the number of nodes in the tree $s$.

The value of $\sigma(T)$ depends on the shape of $T$. Therefore, we will analyse the two extreme shapes that an AVL tree can have: a fully balanced tree (best case) and the Fibonacci tree (worst case). These are the most balanced and most unbalanced shapes that an AVL-tree can have [Knu73].
Balanced trees
A fully balanced tree can be defined as a tree in which every node has either no children or two children that have the same height. In a fully balanced tree of height $h$ there are $2^{h-n}$ sub-trees of height $n$, each of which contains $2^{n+1} - 1$ nodes. Thus in this case:

$$\sigma(T) = \sum_{n=0}^{h} 2^{h-n} c 2^{n+1} - 1$$

Fibonacci trees
Fibonacci trees are the most unbalanced AVL trees. The Fibonacci trees $F_0, F_1, \ldots$ can be defined in the following way:

$F_0$ and $F_1$ are trees that contain only one leaf, and $F_n$ is the tree that has $F_{n-1}$ and $F_{n-2}$ as children.

It can be proved by induction that $|F_n| = 2Fib(n+1) - 1$ where $Fib$ is the Fibonacci function, where the number of internal nodes is $Fib(n+1) - 1$ and the number of leaf nodes is $Fib(n+1)$. The number of Fibonacci sub-trees $F_i$ in a Fibonacci tree $F_n$, for $1 < i < n$, is $Fib(n+1 - i)$. The number of leaf nodes ($F_0$ and $F_1$) in a Fibonacci tree $F_n$ is $Fib(n+1)$.

Thus we can derive the following equation for the number of shared nodes for a Fibonacci tree:

$$s_f(n) = Fib(n+1)c + \sum_{i=2}^{n} Fib(n+1 - i) \cdot c^2 Fib(n+1 - i) - 1$$

This could be further transformed using the identity

$$Fib(n) = \frac{1}{\sqrt{5}} \left( \phi^n - \overline{\phi}^n \right)$$

where $\phi = \frac{1}{2}(1 + \sqrt{5})$ and $\overline{\phi} = \frac{1}{2}(1 - \sqrt{5})$.

Comparison
The behaviour of the modified AVL-tree implementation for versioned sets turns out to be insensitive to both the shape and the size of the AVL-tree.

The proportion of nodes that can be shared with the previous version is independent from the shape of the tree. Figure 4.7 shows an example for a balanced tree (best case) and Fibonacci tree (worst case) of similar sizes.

The proportion of sharable nodes is also independent from the size of the tree. Figure 4.8 shows an example for balanced trees of different sizes.

It can be shown that the proportion of shared nodes is always larger than $1/2c$. As we have seen Fibonacci trees represent the worst case i.e. it gives a lower bound on the number of
Figure 4.7: Influence of tree shape on proportion of shared nodes

Figure 4.8: Influence of tree size on proportion of shared nodes
shared nodes.

\[ s_f(n)/ | E_n | = \frac{Fib(n + 1)c + \sum_{i=2}^{n} Fib(n + 1 - i) \cdot c^{2Fib(n+1)-1}}{2Fib(n + 1) - 1} \]
\[ \geq \frac{Fib(n + 1)c}{2Fib(n + 1) - 1} \]
\[ \geq 1/2c \]

Re-balancing

The algorithm can be further enhanced by performing some extra re-balancing to get a better weight balance. This will make the creation of new sets somewhat slower, but can speed up access to the new set, without incurring additional storage overheads. The enhancement is based on the fact that at certain places in an AVL tree it is possible to re-balance sub-trees without losing the AVL property. These cases are shown in figure 4.9 (the mirror images are not shown). In these cases re-balancing obviously does not disturb the AVL-property: the balance factor\(^2\) for the individual nodes still remains within \([-1, 1]\). However, the weight-balance for these nodes could be improved by this re-balancing operation. If we denote by \(\#X\) the total number of nodes in sub-tree \(X\), and by \(N\) the total number of nodes in the tree, then re-balancing will change the average path length by \(\frac{\#B - \#A}{N}\). Thus, additional re-balancing is advantageous if \(\#B > \#A\).

Re-balancing can be done without causing any storage overhead if the operation is only performed on copied nodes. Thus in our example re-balancing would only be performed if nodes B and D were copied anyway.

A comparison of delta-lists and AVL trees

There are two candidates for the implementation of versioned sets: delta-lists and balanced trees. These data-structures have different properties. The behaviour of the delta-list depends heavily on the value for the parameter \(f\), the proportion of all versions that are stored in full as base version.

If storage space is at a premium, the delta-list with a low value of \(f\) is better than a tree, due to the compact storage that is possible for delta-lists. But the access-time will increase linearly with \((1 - f)\) so low values of \(f\) will give a bad access performance.

If, on the other hand, access-time is more important than storage space, AVL trees will tend to be better, since their access-time is always logarithmic in the size of the set. It seems likely that the delta-list implementation with the same access speed as an AVL tree will consume more storage due to the large proportion of base versions.

Whether or not AVL trees are really superior in this case depends on implementation parameters and usage patterns.

\(^2\)The balance factor of a node is equal to the difference in height between its left and right sub-tree.
4.6 Conclusions

In this thesis (see chapter 2) we have argued that integral version management is an attractive form of version management. Versioning does not come for free, and one of the main reasons that many systems do not provide version management is because it is too expensive. Of course, decreasing hardware costs, increased cpu power, cheaper storage media, such as WORM disks, make version control more attractive. Nevertheless, in order to make large scale application of versioning — and especially integral version management — viable, it is necessary to have suitable implementation techniques.

This chapter presents several data-structures and algorithms to implement integral version management for arbitrary object management systems. They have been used in the
implementation of CAMERA. Although the techniques are applied to versioning of a self-contained object management system, we feel that it is possible to use them also in other situations, e.g. for the versioning of complex objects.

The data-structures described here lead to economical use of disk-space, while giving access-times that are comparable to those of similar systems without version control. Our experiences with the prototype of the CAMERA system are encouraging, and indicate that the frequent creation of new snapshots can be done with an acceptable performance (some earlier work is described in [Lip90]).

Further research will allow us to tune the algorithms to usage patterns. For example, the size of the index for a non-linear history, that was described in section 4.5 depends on the chosen mapping function. Which mapping table performs best is determined by the actual usage patterns, e.g. the similarity between successive snapshots, the rate at which development lines branch and merge etc.
5 Derivation management

5.1 Introduction

In every Software Engineering environment many entities can be seen as the result of the application of a tool to a number of inputs. Familiar examples are invocations of compilers, linkers and parser generators. Derivation management is concerned with modelling and controlling the automatic production of such entities.

This chapter describes the way that derivation management is handled in CAMERA. We start with a short introduction to derivation management. Next we discuss some requirements, and explain why these are desirable. Existing derivation management systems are evaluated with respect to these requirements. The final part of this chapter describes the CAMERA derivation management.

5.2 Derivation management

A good way to look at derivations is to see them as mathematical functions that take a number of inputs and produce a number of outputs (see e.g. [Bor86]). The outputs are completely determined by these inputs. The outputs of a derivation step are called derived values. Derivations can consist of a number of derivation steps, that represent the tool invocations. The results of one derivation step can act as input for another derivation step. A derivation can be seen as the functional composition of a number of derivation steps. We call the graph that describes these input/output connections between derivation steps the derivation graph. Edges in the graph go from the derivation step that produced a specific output to all derivation steps that use this output as one of their inputs.

We distinguish two different kinds of inputs: the first are the derived values, that are the outputs of some derivation step, the second kind are source inputs; these can be changed by the user.

Most derivation steps have a large number of inputs, since all factors that influence the outcome of the results must be modelled as inputs. Some examples of inputs are input files, command line options, as are all other files that are read by the tool. Several other inputs however, are far less obvious. Some tools depend on environment variables. Tools that navigate through directories are influenced by the state of these directories. Furthermore,

\[1\] A variant of this chapter will be published as [LF92].

89
the outcome is of course influenced by the version of the tool that is used, so that the tool itself can be seen as input as well.

When one of the inputs of a derivation step has been changed, this could potentially affect the results of this derivation step. In this case the derivation step must be re-computed to produce the consistent values of the outputs. When these outputs are changed with respect to their previous values, other derivation steps that use these outputs (as their inputs) must be re-computed as well.

An example derivation is shown in figure 5.1. It shows a small example program that uses the yacc parser generator. This derivation consists of four derivation steps. The parser generator reads the file syntax.y and produces y.tab.c. The files prog.c and y.tab.c are compiled by the compiler /usr/bin/cc with the -g debugging option. The file prog.c uses the include-file /usr/include/stdio.h, so that is also an input to the derivation step that computes prog.o. Finally the resulting binaries are linked by the loader /usr/bin/ld using the library /lib/libc.a. This figure is not complete, because some of the inputs are omitted. For example, tools may depend on environment variables, the current directory, and the result of system calls (e.g. to obtain the current date).
5.2.1 Requirements
We now give a list of requirements for a derivation manager, plus a short rationale for their importance. A derivation manager should satisfy the following requirements:

- Maintain consistency between inputs and outputs
- Guarantee reproducible results of derivations
- Support for evolution of the derivation structure
- Be efficient

We will now describe each of these in more detail.

Consistency between inputs and outputs
Consistency between inputs and outputs is the essence of derivation management. When one of the inputs for the derivation has been changed the derived value may have become inconsistent with its inputs. Inconsistencies can lead to searches for bugs that have already been removed from the source code, but not from the binaries, and other waste of development efforts (see [Bab86]). When one of the inputs for a derived value has changed, this derived value normally should be re-computed because it could have been changed, too. In complex systems, it is common that derived values act as input for other derivations. In such systems one small change can therefore result in many out-of-date derived values. Manually tracking which derived values must be re-computed is labour intensive and error prone. It is a task that is best left to a machine.

Under certain circumstances the value may not be re-computed immediately, e.g. because it is too expensive. In this case a warning must be given if the user attempts to use an inconsistent derived value.

Reproducible results
The results of a derivation must be reproducible. At a later point in time we would like to get the same result when the derived value is re-computed using the same inputs. Users are very frustrated if the behaviour of their programs appears to depend on the current phase of the moon. Another reason why reproducibility is desirable has to do with optimisation of storage space. Derived values need not be stored permanently provided they can always be reproduced.

Reproducing a derived value at some later point in time is only possible if all inputs of the derivation are subjected to version control. As we have seen the behaviour of many existing tools depends on several factors. The version management system should control these factors, when these derived values must be reproduced later on.

Managing the evolution of the derivation structure
Normally the structure of derivations is not static. The set of inputs for a derivation step can change when one of its inputs is modified. For example, if a #include <stdio.h>
statement is added to a C-program the stdio.h file must be added as a new input. When new modules are gradually added to a program during program development new inputs must be added to the derivation step that represents the compilation of the program. The computations that are used in derivations may also change. Initial versions of a program module will be compiled with a debugging version of the compiler while more mature versions will be compiled with an optimising compiler.

The tools that are used in derivation steps may change when new releases become available. The derived values that have been computed using this tool are affected by this change. The derivation manager must adapt itself easily to such changes. It must re-compute the derived value and start monitoring the new set of input objects.

Derivation structures can grow large and become very complex. Multi-step derivations, where the results of one derivation step are used as input to the next one, are very common. In large derivations it becomes difficult to comprehend the entire derivation structure. The derivation manager should help the user managing this structure, by being able to support queries about the derivation structure.

One type of queries asks for the total set of input dependencies for a given derivation step. Inputs for a specific derivation step may have been derived values themselves. Therefore, a transitive closure must be computed to determine all non-derived values that act as input for a given derivation step. This type of input dependency information is useful for determining which objects are part of a given product. When the product must be transmitted to another place it is important to have this information.

Information about the derivation structure can also be used for impact analysis: Which derived values could be affected when a given object is changed? This information is used in deciding whether a given modification should be performed. When many derived values depend on this object it may be too expensive to change it. Furthermore, changing one input in a derivation may necessitate changes in other inputs as well. For example, when the interface a procedure in one module is changed, other modules that use this procedure may also have to be updated. These modules can be found by using the derivation graph.

**Efficiency**

The derivation manager should be efficient: it should not consume large amounts of computer resources.

One way to be efficient is to prevent unnecessary computations. A very naive method to maintain consistency between derived values and their inputs is to re-compute the derived value every time that the user requests its value. But derivation steps can be very costly, and recomputing all derivation steps every time will be very expensive for large derivations. The derivation manager should maintain a cache of derived values. The cached value can be used if it can be determined that it is still up-to-date.

The computations that are needed to determine whether a derived value is still up-to-date must also be efficient themselves.
5.3 Existing systems

Several systems have facilities for derivation management. Some of these systems will be
introduced and examined to see how well they meet the requirements.

The archetypical example of a derivation manager is Make [Fel79] (see section 1.2.3). Make
uses a description of a derivation that is stored in a Makefile. A Makefile for the
element in figure 5.1 is shown in figure 5.2. Make defines how files can be produced from
other files. For each file a list is given of the files on which it depends, and furthermore a
command is given that will produce this file. This Makefile is slightly unusual in that we
have attempted to capture dependencies on system supplied tools, libraries and include
files. Frequently the assumption is made that these are constant and are not subject to
change. Another frequently omitted dependency is that on the Makefile itself. The most
important reason for this is that otherwise all derivations will be re-computed, when the
Makefile is changed. Thus for example if the –g option is removed from the Makefile
for the compilation of prog.c the file y.tab.c will be built even though the result is
not influenced by the modification to the Makefile. However, if we would have omitted
the dependencies on the Makefile, prog.c will not be recompiled automatically.

The popularity of Make has paved the way for many Make-derivatives, such as [Hum84,
EP84, Baa89, ML89] which extend the basic functionality to deal with issues such as
parallel computation and distribution. Make is also the basic derivation manager in more
recent systems for configuration and derivation management, such as Shape [ML89],
Imake [Ful89] and NSE [NSE88, CFH88]. A notable different approach to derivation
management is the Odin system [Cle86, Cle88a], while a more formal treatment of the
derivation process is given in [Bor86]. DSEE [LC84] has a “Configuration Manager” that
handles derivations.

Some existing OMSes also handle derived information. An example is the Cactis system
[HK88] that can handle derived relations.

These existing systems can only partially satisfy our requirements.
Consistency between inputs and outputs

All existing systems will under certain circumstances fail to maintain consistency between derived values and their inputs.

The first situation when derivation managers will fail, is when some input dependencies for a derivation step are not made explicit. In this case it is not possible to guarantee complete consistency. When one of these “hidden” inputs is changed, the derivation will not be re-computed, since the derivation manager is not able to detect this situation. In Make and its derivatives users must explicitly specify the derivation structure. It is easy to forget some inputs when this is done manually. Existing Makefiles frequently omit dependencies. Tools like makedepend can be used to analyse the include-files for a given C program, and to add these to the Makefile. These tools will only detect dependencies for one specific language, and they will not detect other dependencies, e.g. on the version of the compiler or other tools, or dependency on environment variables. Furthermore, they only solve the problems when they are invoked frequently. A slightly different approach is used by some Makes, that record the files that are actually included during each compilation (using a modified version of the compiler). These dependencies are recorded in a “makestate” file that is maintained by Make. This approach is somewhat better than the makedepend approach because the dependencies are automatically updated during each Make run. However, it still has not solved the other problems: language dependency and omission of dependencies.

Derivation managers will also fail to guarantee consistency if it cannot be determined that an input has changed. Many derivation managers use a time-stamp technique to determine whether an input (file) has changed. However, there are some inputs that are notoriously difficult because they are not represented in the file system. A famous example are the Unix environment variables. The value of an environment variable can be different for each process. Another example of uncontrollable inputs are system calls that are performed by tools, e.g. to obtain the current directory or the current time. Odin solves the problem by disallowing environment variables. All derivation computations are run from a parent process that explicitly removes all environment variables. In a similar way dependencies on directories are partially prohibited by running all derivations in a directory that is controlled by Odin.

A third case where derivation managers will fail, is when users can access derived values without intervention of the derivation manager. In Make-like systems the user can immediately access the derived values since these are represented as files in the user’s workspace. Re-computations of derived values are not done automatically, the derivation manager must be explicitly invoked. With such derivation managers it is very well possible that out-of-date outputs are used, because the derivation manager will not automatically give a warning when this happens. Systems like Odin avoid this problem since the derived value is not represented as a file in the user-directory. The only way in which a user can get at a derived value is through the Odin system.
Reproducibility

Most derivation managers are not very well integrated with version management tools. This makes it difficult to guarantee reproducibility of derived values.

One factor that hinders reproducibility is the fact that it can be difficult to apply version control to all inputs of a derivation. For example, the behaviour of many tools depends on directory structures, which however are frequently not subjected to version control. The tools that are used in derivation steps must also be version controlled if derivations must be reproducible. Simple derivation systems do not record the versions of the tools. More advanced systems, like DSEE, do attempt to record the version-identification of the tools.

A further integration problem is that most version management systems can only handle text files. They cannot be used to record versions of non-textual inputs. For example, they cannot easily be applied to subroutine libraries.

Another integration problem between derivation management and version management is the selection of versions of inputs. When a version of an input item must be used that is not the current (e.g. checked-out) version, this version must be checked-out before the derivation can be performed.

Attempts have been made to obtain a better integration. For example, some Makes attempt to check-out a file from SCCS when it cannot be found. But these facilities are still rudimentary, and have a very ad hoc character.

Systems where derivations can have side-effects on other derivations have problems with reproducibility. In such a case one derivation step may influence others. An example of this is \LaTeX{} [Lam86]. \LaTeX{} behaves as a multi-pass compiler where the results of one run are written on auxiliary files and used as input in the following run. \LaTeX{} uses this facility for handling page references, the table of contents, and bibliography information. In this case the results depend on the number of times \LaTeX{} was invoked. The number of times that a specific step was executed can influence the outcome of derivations. This is undesirable because the computation of a specific derivation is no longer invisible. In Make derivatives the derivation steps can perform arbitrary actions on the file-system (thus with side-effects). Odin attempts to prevent side-effects. Derivation steps are executed in a separate directory so the results of one derivation cannot have effects on other derivations.

Managing the evolution of the derivation structure

During development the structure of derivations will change frequently. New derivation steps will be added, and existing ones will be changed. Most derivation managers cannot deduce the derivations from the objects themselves, but use a separate description of the derivation. When the objects are modified it can be necessary to update this derivation description. For example, if an include statement is added to a source file, the corresponding input must be added to the derivation step in which this source file is used. When the derivation manager cannot detect this additional input automatically it must be added manually to the description of the derivation. It is easy to forget updating the derivation
description when the derivation structure is changed. Tools have been made that automatically construct a part of a derivation description, but these only work if they are invoked regularly.

Most derivation managers offer only limited support in managing large derivations.

In Make-like systems it can be difficult to determine where the derivation description for a specific derived value can be found. This is especially difficult when multiple directories are involved. A Makefile in one directory may describe the derivation of a file in another directory. Thus it can be difficult to determine whether some file represents a derived value. A factor that adds to these complications is the use of program generators (e.g. parser generators) that produce a piece of source text as output. In this case it may happen that these derived values are edited. Of course these modifications will be lost when the derived value is re-computed. In DSEE and Odin this problem is avoided because the name of the object describes how it was constructed. These systems maintain a separate name spaces for derived values. Thus there is a clear distinction between real inputs and derived values.

Existing derivation managers do not provide much further information for dependency analysis or impact analysis. NSE can generate a list of input files that were used to produce a given Make target, using the "makestate". But none of the derivation managers can easily give lists of all derived values that depend on a given input.

Another problem with several derivation managers is that the specification of derivations can grow very large. Many derivations show regular patterns. For example, the compilation of a program commonly consists of compiling all modules in the program and then linking them together with some libraries. With Make repetitions of very similar rules across multiple Makefiles cannot be avoided. Some standard rules, e.g. for compiling and linking C-programs are built-in. Other rules must be added by the user and must be repeated again and again in Makefiles. Imake tries to alleviate this problem by providing templates for certain frequently re-occurring derivations. Odin uses the types of objects and information about tools for describing derivations. Each object in Odin has a type that determines the operations (i.e. the tools) that can be performed on this object. When an operation is performed on an object that is not defined for its type, Odin attempts to convert the object to a type for which this operation is valid. The advantage of this approach is that derivations need not be respecified everywhere. This approach leads to easier specification of derivations and to easier re-use of similar derivation steps.

Efficiency

Most existing systems are reasonably efficient. They use a lazy re-computation strategy: computations are only performed when the user requests a derived value that is out-of-date. A slight disadvantage of this scheme is that the user may have to wait a while until all computations have been performed. With an eager recompilation scheme, all derivations are immediately re-computed whenever one of their inputs has changed. This is much more expensive.
Two different techniques are used to determine whether an input has changed. The time-stamp technique uses the time-stamps of the input and output files of a derivation step. A derived value is up-to-date if it is newer than all its inputs. Make derivatives are based on time-stamping. The other technique to determine whether an input has changed is to compare the old and the new value of an input to see whether they are different. The value comparison techniques can avoid some rederivations for multi-step derivations. It is well possible that a change in the inputs for a specific derivation step does not affect the value of the outputs of that step. Other derivation steps that use this derived value as input, will always be re-computed with time-stamping while with value comparisons they will only be re-computed when the value has actually changed.

In some cases it can be decided that re-computation is not necessary (because results would not have changed) without performing the derivation itself. A famous example is the changing of comments in a program source. Even though the input has changed the compiled program will still be the same. When the derivation manager can determine cheaply that output values will be the same, the derivations need not be re-computed. A technique to avoid unnecessary re-computations is described under the name “smart recompilation” in [Tic86]. This technique however is very dependent on the semantics of the derivation, e.g. on the programming language that was used. Similar work has been done in the context of incremental evaluation of attribute grammars by Reps et al [RT89].

5.4 Derivation management in the CAMERA OMS

The remainder of this chapter discusses the model for derivation management that has been integrated into the CAMERA Object Management System. The essence of this approach is that derivation management is an inherent part of the OMS itself. This differs fundamentally from the more traditional situation in which a separate derivation management tool operates on a passive object store (like a file-system).

The derivation management in the CAMERA OMS may be described as follows. Within the OMS, derivation steps are modelled as functions that operate on the contents of the OMS. For each of the derived values the derivation manager maintains a dependency set, which describes all the elements in the OMS on which the computed value actually depends. Since the dependency set is maintained by the OMS implementation, users need not indicate dependencies explicitly.

Dependency maintenance for a derived value is not limited to “data dependencies”, e.g. when the contents of a file is used in the computation of another object. It also detects navigational or structural dependencies, e.g. when a derived value depends on the existence of a particular directory entry. The total dependency set for all derived values is represented in the OMS itself, and can be used for new applications such as impact analysis. The re-computation strategy to be used for a particular derived value, e.g. eager or on-demand, can be indicated by the user. The manager maintains a cache of derived values.

In the remainder of this chapter the architecture of the CAMERA derivation manager is described. This discussion is partitioned into sections, each of which addresses a particular
design aspect of derivation management and compares our approach to others. Section 5.4.1 discusses how derivation steps, tool inputs and tool outputs are represented in the OMS. In section 5.5 our approach to dependency maintenance is described. This section also illustrates some non-traditional applications of derivation management that are possible in our system. Section 5.7 discusses the issue of re-computation and cache maintenance. Finally, section 5.8 provides some conclusions and future steps.

5.4.1 Representing derivations

In the CAMERA OMS tools are considered to have functional behaviour and therefore derivations are modelled as functional methods that are associated with classes. For example, a compiler is considered as a function that takes a source text as input (plus possibly a set of options) and that returns the corresponding object code. Functions can return large and compound values. This also captures tools that have multiple return-values. By treating derived values as functions of its inputs CAMERA is very similar to Odin [Cle88a, Cle86] and to the work by Borrison [Bor86].

Using methods to describe derivations has as additional advantage that all objects of the same class can share the same method. In Make, derivations are usually described at an instance level, and it is therefore common that similar derivations are repeated across different Makefiles. This can cause problems in large systems which consist of hierarchies of Makefiles. If a particular kind of derivation step (e.g. the tool to invoke when compiling a C-file into an object module) should be altered across these files, it may be easy to forget one or two of the instances of the kind. Consequently, several mechanisms, such as "built-in derivation rules" or Make (environment) variables have been introduced to address this issue. On a larger scale, lmake [Ful89] generates full Makefiles from simple derivation descriptions, and a system specific set of tool descriptions and derivation rules. The Odin system [Cle86, Cle88a] also addresses this issue. Odin uses a description of the available tools, their inputs and outputs and the options. Based on this specification, a user can obtain particular derived values based on the available source values. We have considered using the Odin language for the specification of derivations in CAMERA. Odin has some interesting features, that are not available in CAMERA, e.g. users don't need specify intermediate steps, since Odin will deduce which intermediate operations are needed. We decided not to use Odin because it would have duplicated many of the CAMERA derivation management facilities, and would confuse the user when two similar but different mechanisms were available.

Within our approach options are handled as additional inputs. It is therefore possible to put options in a separate object, that can be shared by many derivations. Treating options like inputs is essential for maintaining the consistency between inputs and derived values (see [Bor86]). In addition, it also contributes to a uniform representation of the derivation process.

Uniformity is also increased by modelling derivation as methods. From the user's perspective there is hardly any difference between a derived value and an ordinary method.
Furthermore, the description of how a derived value is produced is directly represented in the function itself, and not in some separate Makefile.

Because derived values are implemented by functions without side-effects, they can examine arbitrary parts of the contents of the OMS but cannot change its state. This has the advantage that derived values need not be version-controlled, since they can always be re-computed from inputs. Since computations are guaranteed to have no side-effects, different derivations cannot interfere with each other, and thus the computed derived value is independent from the order in which the computations are performed. The cache manager (see section 5.7) can therefore perform arbitrary optimisations (e.g. throw away intermediates or throw away cache entries that are rarely used)

Many tools create and delete temporary files, which have side-effects on the file-system. In our approach derivations cannot use the OMS to store these temporary entities, because functions cannot have side-effects. Temporary files are mainly used for communication between different programs (e.g. different passes of a compiler). If these programs can be modelled as functions that return the contents of these intermediate files, there is no need to have side-effects on the OMS, and the tool can be implemented in a functional manner.

In our model derived entities are clearly different from normal (stored) entities. In traditional systems it is hard to distinguish between files that are simply stored, and files that are the results of a derivation. This can lead to problems, for example when users start editing intermediate derived files (e.g. editing yacc output) that will be overwritten in the next derivation. In these systems it is possible that objects that are the result of a derivation, disappear. This introduces problems with referential integrity: if a derived object disappears, because one of its inputs was changed, the question is what should happen with references to this object. The reverse problem exists as well: how can we get a reference to an object that is newly created, due to a change in the inputs for a derivation.

5.4.2 Example
This section introduces a small (simplified) example to illustrate our approach. The example is loosely based on the use of the \TeX\ ([Knu84]) document formatter. This example will also be used in section 5.5.2 as an illustration of dependency maintenance.

```plaintext
Class TeXSource Superclasses edit{
  TeXSource Function dvi (): string
  ...
  TeXSource Function get-include (string): string
  ...
  Relation TeXMacroDir: TeXSource ↔ integer ↔ Object
  Relation directory: Object ↔ symbol ↔ Object
  dir-exists Function Object (symbol): boolean
  ...
```
Objects of the class TeXSource contain the \TeX{} input in their attribute value. This attribute is inherited from the class Edit that was described in section 3.1.2. The corresponding dvi-file is treated as a derived value and is thus represented by the function dvi. A \TeX{}-file can contain references to other \TeX{} files, and these are found by searching in a set of directories. Every TeXSource is related to the corresponding directories by the relation TeXMacroDir. The relation TeXMacroDir is a ternary relation. The first column represent the TeXSource and the last column represents a directory object (in \textsc{camera} any object can act as a directory object; directory entries are recorded in the Directory relation). The second column indicates the order in which the directories must be searched. Directories with lower numbers are searched before the directories with higher numbers. The function get-include takes the name of an include-file as argument and returns its contents as a string. It locates the include-file object that corresponds to the given name by examining the directories in the order indicated by TeXMacroDir. The function get-include is also a derived value. The relation directory is analogous to the directory in traditional file-systems, the entries are 3-tuples (directory, object-name, object).

5.5 Dependency maintenance

A major issue for derivation managers is how the set of dependencies for derived values is maintained. The dependency set indicates the values on which the result of the particular derivation step depends.

In the previous section we have seen how derivation management has thus far been achieved by building on top of existing file-systems and modifying existing tools in a rather ad hoc manner. One of the main problems with these tools was the determination of the input sets for a derivation step.

In the \textsc{camera} system, it was decided to incorporate derivation management in the OMS itself, instead of treating it as an add-on facility. As a consequence users need not specify or maintain dependency information. The object management system maintains a dependency set for all derived values that occur in the OMS.

This dependency set is not hidden. It is represented as a relation in the system and can be exploited in other tools, such as impact analysers.

5.5.1 The dependency set

A few definitions are needed in order to describe how dependency management works. We define the immediate dependency set of a derived value as the set of inputs that was used to compute that derived value. In the \textsc{camera} OMS this set contains object attributes and derived values. Since an input can also be a derived value itself, we can also define the complete dependency set with the following recursive definition: a complete dependency set of a derived value is the union of its immediate dependency set and the complete dependency sets of all derived values in this set. Observe that the dependency set for a
derived value contains all parts of the OMS that are needed for the computation of this derived value. Consequently, a derived value can only change if one of the inputs in its dependency set changes. Note also that the dependency set itself can change if the value of one of its members changes. Take for example the case where we change the inclusion of "a4.sty" to "a3.sty". Now the include-file denoted by "a4.sty" is no longer member of the dependency set, but the one denoted by "a3.sty" is.

It is obvious that the derivation manager must be able to determine the dependency set for each derived value, since otherwise it cannot detect that an derived value is no longer up-to-date. The derivation manager determines this dependency set with a surprisingly simple method: it monitors which parts of the OMS are examined during the computation of a derived value. Since it is possible to trap all calls by the tools to the OMS, the dependency set can be maintained for any tool, without a need for special dependency analysers. Furthermore, this approach guarantees that dependency sets are complete and up-to-date: it is impossible to overlook a certain dependency by accident. Consequently, the derivation manager will notice all cases when a derived value needs re-computation, and is therefore guaranteed to provide the actual dependencies.

Since the information about the dependency sets for each derived value are maintained anyhow, it is not a hard job to make this information also available to the user. The CAMERA OMS presents this information in the form of a read-only relation, called the depends-on relation, that relates every derived value with its immediate inputs. In CAMERA a derived value is the result of invoking a function on an object. Thus derived values can be represented as a 3-tuple ((oid, method-selector, args)), where oid is the object to which the message is sent, method-selector is the selector (the name) of the function, and args is a list of arguments. A derived value can depend on other derived values, and on the values of some attributes. Each dependency is represented as one tuple in the depends-on relation. For dependency on derived values this tuple looks like:

<oid1, sel1, args1, attribute, oid2, attribute-name>
<oid1, sel1, args1, derived, oid2, sel2, args2>

Where {oid1, sel1, args1} describes the derived value. The derived value can depend on attribute attribute-name of object oid2 or on derived value {oid2, sel2, args2}.

5.5.2 Example

The following example illustrates some of the properties of the approach, based on the example classes from section 5.4.2 (see figure 5.3).

In this example we have a TeXSource object, paper.tex. Furthermore, there are two directories (localMacros and standardMacros), respectively containing the locally developed macros and the standard \TeX macros. Both directories are related to paper.tex by the TeXMacroDir relation. localMacros has a lower integer in
its tuple than standardMacros, thus it will be searched first. As indicated earlier the
dvi-value is modelled as the function dvi on object paper.tex. This dvi method
invokes the method value to obtain the contents of paper.tex. In turn the method
value depends on the value attribute of paper.tex. For finding the correspond-
ing included files the function dvi calls function get-include. Let us suppose that
paper.tex includes a file called a4.sty. Then the function dvi will call the func-
tion get-include(a4.sty) on paper.tex. The result of this function can also be
 treated as a derived value. The function get-include queries the TeXMacroDir rela-
tion, and finds the two directories localMacros and standardMacros respectively.
The function get-include will first search localMacros. If localMacros does
not contain a macro file a4.sty, it will search standardMacros. The relevant part of
the depends-on relation is shown in figure 5.4.

As can be seen from this example, the system can detect dependencies on relationships.
Thus if a relationship is changed (for example if a different TeXMacroDir is changed)
this will be detected by the system. A further property is that derived values can also
depend on objects that do not yet exist. In our example, if a macro file a4.sty is added to
localMacros, the dependency set for (paper.tex, get-include(a4.sty)) will be different since there is now no need to search standardMacros. The
depends-on relation will be automatically adapted to this situation.

It should be noted that intermediate derived values can act as fire-walls that guard against
unnecessary re-computations. In our example all kinds of files can be added to both
macro directories, but they will only trigger a re-computation of (paper.tex, dvi),

Figure 5.3: An example TeX configuration
if they are named a4 sty. In the same way we can change the relation TeXMacroDir in any way we like as long as the result of \{TeXMacroDir query \{(paper.tex * *)\}\} remains the same. Furthermore, the dvi file will also not be re-computed when a copy of a4 sty is added to local Macros because the value of \{paper.tex get-include \{a4 sty\}\} remains the same.

5.5.3 Applications of dependency management

We believe that the dependency sets constitute valuable information. This section describes some ways in which dependency information can support the user, and it describes some of the applications of dependency information as it is used in the CAMERA system. Explicitly represented derivation information is a good basis for the analysis of derived values within a complex derivation process, since it is guaranteed to be complete. This can be important information for release management. When producing new releases, it is essential to record the current version for all inputs that are used to construct the released product. Obviously, the depends-on relation can be used to determine this in a very straightforward way. The depends-on relation can also be used the other way round, i.e. to do impact analysis. Before some input is changed it can be useful to know which derived values depend on this input, and thus might be affected. For example, it can be used to determine whether a certain include-file is used and to examine what the consequences could be of changing this include-file.

As a final example we consider the use of the dependency set in determining related groups of objects. The CAMERA OMS supports the notion of pieces, sets of related objects together with relationships (see chapter 6). Pieces are used as a communication mechanism: users exchange pieces to share developments. The dependency relation provides very useful information for the piece editor. If a piece contains a derived value, the receiver of
the piece should be able to supply all inputs for this piece. If it cannot be expected that
the receiver already possesses a particular input, it should be included in the piece, so it is
exchanged together with the derived value itself.

5.6 Equivalence of derived values

For certain purposes, different derived values can be considered to be equivalent. For
example, an object code module that was compiled with a debugging flag might be re-
used at places where a "normal" version of this binary was required, since the debugging
flag is not considered to change the behaviour in any important aspect. So when the
computation of a derived value is expensive, it is advantageous to use an "equivalent"
derived value if such a value can be found in the cache. Some derivation managers (e.g.
DSEE and Odin) have implemented such a notion of equivalence. In general they ignore
certain options. The user can state that the value of the debugging flag is not important,
and thus the versions with and without the debugging flag are considered equivalent.

The decision whether two different derived values can be treated as equivalent not only
depends on the versions themselves, but also on the purpose for which the derived value
will be used. For example, many compilers have an option to include debugging informa-
tion in the produced binary, and an option to optimise the produced code. These options
are frequently mutually exclusive. Thus if we want to debug a specific module we would
prefer a version that has been compiled with the debugging flag on. Otherwise, during
development it is possible to see a binary with debugging information as equivalent to a
binary that was compiled with the optimising flag. However, for the final production ver-
sion of the program these two binaries are not equivalent, because the optimised version is
preferred. Thus whether two derived values are equivalent does not depend on the values
themselves, but also on the purpose for which they are used. Existing derivation managers
do not appear to support this notion of equivalence directly. Odin does have a mechanism
for sub-typing that could be used to model such dependencies. For example, we could use
types debugo, opto that are produced by the compiler with debug and optimise flags
respectively. Both could be declared as subtype of type o that would contain all compiled
files. Users could then use the correct files by selecting the correct types.

Consequently, it is impossible to define a general procedure for determining equivalence.
Instead, our approach is to let the users define their own form of equivalence handling,
by allowing them to interrogate the cache manager. Users can ask for one of a set of
derived values, which they consider equivalent. This is done with the scheme-function
select-derived-value. This function takes one argument, a list of derived value
specifications, each of which is a list (old, method-selector, args). The cache
manager will return one of these values if it is currently cached, or will re-compute one of
these otherwise.

The advantage of this approach is that equivalence is determined at the place where the
derived value is used. A disadvantage of this approach is that the contents of the cache
will have an influence on the derived values, which hinders reproducibility. Therefore, this feature should be used carefully.

5.7 Cache management

The cache manager is responsible for maintaining a cache of derived values. As mentioned earlier, the cache manager is notified whenever some part of the OMS is changed, and can then determine whether any derived values need to be re-computed. Because the cache manager is integrated with the OMS, techniques like using time-stamps to detect modifications (as done by Make) are not necessary.

The cache manager will re-compute a derived value if the value of any of its inputs has changed. In multi-stage derivations this could give a better performance than time-stamp based methods. Time-stamp based methods will always re-compute all derived values when a direct or indirect input has changed. Thus for example if a source file is modified, the compiler will be invoked to produce a binary, and then the loader will always be invoked to re-link this binary. In our approach the loader will only be invoked when the value of the binary has really changed; thus a change of the comments in the source file will not lead to an invocation of the loader, since the system can detect that the binary has not been changed.

When the cache manager detects that an input for a derived value has changed it can follow two different strategies to re-compute the derived value. In the lazy strategy nothing is done when an input is changed. However, when the user requests a derived value whose inputs have changed since the last computation, this value is automatically computed, and the new, up-to-date, version is returned to the user. In the eager strategy all derived values are immediately re-computed as soon as inputs are changed. Observe that both strategies guarantee that the user always accesses an up-to-date derived value. They differ only in the point in time when the values are re-computed.

Since all derived values can be re-computed at any time, the cache is truly a cache, and therefore the cache manager has considerable liberty in deciding which values must remain in the cache and which ones are deleted, e.g. due to lack of storage space. The user can give hints to the cache manager to improve its performance, e.g. which values can be deleted from the cache because they are not needed in the near future, and which values must be certainly be retained because it is expected that they will be used intensively.

The algorithm used by the cache manager works as follows. The cache manager maintains the set $\mathcal{R}$ of all derived values that must be re-computed because at least one of their immediate inputs has changed. The system maintains the dependency relation $D$, described here as a function $D(x)$ that returns for a derived value $x$ the set of all its immediate inputs. In a similar way $D^+$ and $D^*$ describe the transitive and transitive-reflexive closure of $D$, e.g. $D^+(x)$ is the set of all (immediate or transitive) inputs for $x$, and $D^*(x) = D^+(x) \cup \{x\}$.

The function $\text{eval}(x)$ returns for a specific derived value $x$, its current value plus its dependency set. The entry point in the algorithm is the function $\text{get}(d)$ that returns the current value of the derived value $d$. 
get(d)
    while D*(d) ∩ R ≠ ∅ do
        select s ∈ D*(d) ∩ R such that D*(s) ∩ R = ∅
        recompute(s)
    od
    return cachedval(d)

recompute(z)
    R := R − {z}
    (newval, D(z)) := eval(z)
    if newval ≠ cachedval(z) then
        changed(z)
        cachedval(z) := newval
    fi

changed(z)
    R := R ∪ D−1(z)

5.8 Summary and open issues

In this chapter we have described a model in which derivation management is integrated with an OMS. Incorporating these facilities at a basic level offers important advantages over the traditional approach where derivation management is left to separate tools. The main characteristics of our approach are:

- Derived values are modelled as functions on objects and are thus cleanly integrated with the rest of the OMS.
- Consistency between derived values and inputs is guaranteed.
- Derived values can always be reproduced.
- The derivation manager supports the evolution of derivations. Dependency information is recorded and maintained automatically and in a tool independent way. Users do not have to specify any dependency information, because it is automatically deduced from the behaviour of the tools.
- Derived values can depend on arbitrary elements of the OMS. Dependencies are not limited to objects, but can also include dependencies on structural information such as relationships, or the non-existence of certain objects.
- Explicit representation of dependency information helps the user managing the structure of the derivations. It can be used to perform dependency analysis and impact analysis.
• Existing Unix tools can be incorporated (see section 8).

A prototype of the CAMERA OMS, that includes the dependency management facilities described in this chapter, has been implemented.

5.8.1 Open issues

In some cases, the inherent consistency between derived values and their inputs might cause problems (see [SK88]). When re-computation of a derived value is very expensive, situations can occur in which developers prefer to use an old version of the derived value while changing its inputs. Another case in which (temporary) inconsistency might be desirable is to control the propagation of errors. If, during the derivation of a complex application, one compilation fails, the entire derivation will fail, inhibiting any further development or testing until this error is repaired. A similar case occurs when tools are represented as derived values, and development of the tool proceeds while the tool is in active use. New untested versions of the tool are derived automatically. Since these new versions might contain bugs, users would prefer to continue using the old and trusted version. In CAMERA these problems are less severe than in most other systems. First of all, environments are isolated from one another, so these problems only influence the environment of the developer that is modifying the tool itself. Second, when serious errors are made it is possible to undo the effects of the erroneous modification by using the old version from a previous snapshot.

There are two obvious approaches for handling these issues. A straightforward solution, which fits directly into our model, is to copy the current value of a derived value into an object and to use this copy. Of course, this only avoids unnecessary, expensive recomputations if a lazy strategy is used. A more radical solution would be to extend our model of derivation management, so that it is possible to freeze the current state of a derived value, and explicitly allow it to be (temporarily) inconsistent with its inputs.
6 Pieces

6.1 Introduction

Good mechanisms for exchanging data among users are needed for any form of cooperative work. In CAMERA this need is even more pressing due to the loosely-coupled nature of the system. One of the mechanisms for exchange in CAMERA is the exchange of entire snapshots. This is not always a suitable mechanism because by using entire snapshots it is not possible to integrate the results of multiple users. An exchange mechanism that allows the exchange of parts of a snapshot is needed for this integration.

In very simple cases users only want to exchange individual objects. But in many cases exchanging single objects is not sufficient. In software engineering projects developers work on different sub-systems of a larger program. Often entire sub-systems are communicated where each sub-system consists of multiple objects, e.g. multiple source files. Similarly, a tool usually needs several auxiliary objects, e.g. its documentation and library files. When (a new version of) the tool is transmitted, new versions of auxiliary objects may be included too.

Exchange of multiple objects occurs more frequently when the size of objects decreases, because individual objects do not contain sufficient information. The exchange of multiple objects also becomes a necessity when the frequency of contacts decreases, since the participants have performed more work and must therefore exchange more data.

Finally, when the set of objects grows it becomes unacceptable to have to select all objects that must be transmitted manually. Automated support to make this selection becomes necessary. This chapter starts by listing requirements for an exchange mechanism, which are then compared with existing exchange mechanisms. CAMERA supports a notion of exchangeable groups named pieces. The rest of this chapter describes pieces and their implementation. Pieces are constructed by using a special tool called the piece editor that is the subject of the final part of this chapter.

6.2 Requirements

We start by listing some requirements for a mechanism that supports the communication of related groups of objects and relationships.
Completeness of pieces

In OMSes objects can be highly inter-related. Objects can have many interdependencies. An object that contains the \TeX source for some documentation normally has multiple inputs, style files and fonts that all contribute to the printed output. A similar structure is seen in large programs that use include-files, libraries, program generators, derivation managers, etc. Manually adding all relevant objects to a piece is infeasible. The system should keep track of such dependencies on other objects and warn when they are omitted from the piece.

The contents of a piece has to evolve over time, e.g. due to the creation, modification or deletion of objects. This should be detected and the piece must be adapted accordingly.

Support for repeated exchanges and ad hoc exchanges

In some cases it can be planned beforehand that the same group of objects will be exchanged on a regular basis. When one user is responsible for the maintenance of a certain program, it can be planned that a piece must be created that contains the program, together with auxiliary objects that are associated with this program, such as documentation, libraries and sources. This piece is sent to customers of the program when new versions are created.

However, other communications have an occasional character: a user may be interested in some objects that have been created or modified by someone else. E.g. someone has drawn some nice pictures, or wrote some useful shell scripts. This kind of exchanges can be ad hoc: the original user may not have expected that these objects would be transmitted, and this might be a one-time event.

Pieces must not force a specific organisation of data on the user

The definition of a new piece should not force the user to re-structure existing objects. Applications impose certain requirements on the objects and relationships in the OMS. Each application can have its own notions of which sets of objects forms a conceptual unit. Because CAMERA is intended as a general purpose system it must be prepared to handle a wide range of applications. Therefore, the piece mechanism must be able to cope with a wide range of structuring conventions. It must be possible to combine an arbitrary set of objects and relationships in a piece, without having to modify the objects or their relations.

Support for receiving

Objects in an OMS may have many relationships that are needed for proper functioning of these objects. For an object in a specific piece these relationships can be internal, when they refer to objects that are also in the piece, or external, when they refer to objects that are not in this piece. When a piece is received the external relationships must be re-established so that the objects in the piece can function correctly in the new environment. An object in a piece that contains a program source text normally needs a compiler. It is not always
useful to include the compiler itself in the piece because it may not function at all in the receiving snapshot. When the piece is received the relationships must be established with an appropriate compiler.

6.3 Existing approaches

In file-based systems, archive programs such as \texttt{tar} are frequently used for the exchange of a set of related files. An archiver takes a set of files and stores their contents in another file, the archive file. The archiver can then extract files from the archive file and restore them to their original state. Many different archivers exist. In the Unix domain, \texttt{tar} and \texttt{cpio} are widely used. Other archivers are mainly used on micro-computers, e.g. \texttt{arc} its many derivatives, and \texttt{200}. The last group also compresses the contents of its archive to reduce storage requirements. On larger computers this is frequently done by a separate program. Most archivers do not only record individual file-names but are also able to store directory information. Because there are no general tools available to generate a list of files and directories that must be stored in the archive, frequently these tools are only used to store entire directories. Thus the contents of the piece is normally only described in the command script that is used to generate the archive (e.g. in a Makefile).

A somewhat more advanced mechanism is to group related objects into \textit{layers}. Layers have a total ordering imposed on them, a layer is either above or below another layer. The same object can be defined in multiple layers. The value that is actually used is the one from the highest layer in which it is defined. Thus redefinitions in higher layers completely hide definitions in the lower ones. In most systems a layer can also contain a special deletion mark for an object. This makes the object definition entirely transparent to the higher layers.

Layers can be shared among groups of people. Normally the lower layers are shared by a large group, while the higher layers contain the local modifications with respect to the lower layers. The modifications by a user are added to the top-most layer, that is private to this individual. When objects in a layer have become sufficiently stable they will be installed in lower layers, and are then shared by larger groups. Layers can be used to group a related set of objects for communication.

Several systems use a layer-like approach. The PIE system [GB86] provides a layer mechanism for Smalltalk classes and methods. A similar system for use in a hypertext system is described by Prevelakis [Pre90]. The Translucent File System [Hen88] provides layers of entire directories on Unix file-systems. The DaSC system [GMS89] provides a layer mechanism that is built on top of traditional file-systems. This system has been developed for the construction of real-time software in a very heterogeneous environment. One application of layers in DaSC and TFS is as a variant mechanism on entire file-systems, e.g. to specify which files/directories are necessary for a given variant of the product.

In general a layering approach has the advantage over the archiver approach that layers are recorded persistently, and that objects in one layer need not be stored in the same
directory. A disadvantage of layers is that it may be difficult to determine the effects of interactions among layers. Two objects in the same layer cannot depend on the fact that they will be used together because one of them could be replaced in a higher layer. Thus the effect that a layer has depends on the layers below and above. These interactions among layers can cause unexpected effects when layers are exchanged.

The Epoch mechanism in KBSA that was described in section 1.2.13 can also be seen as a mechanism for grouping and exchanging related objects. An Epoch can be seen as a versioned set of multiple objects. Users can determine which version of an Epoch they want to use. They can use the modifications that were made to an Epoch by selecting the version of the Epoch that contains those modifications. Epochs only support exchanges between users on the same (non-distributed) Object Management System.

Composite objects are used in several systems (e.g. Orion [BCG+87] and PCTE [ECM90]). In these systems certain references among objects are treated as composition references. A composite object is the set of objects that are reachable from an object via the transitive closure of these composition references. Composite objects in Orion and PCTE can be versioned, and each version in itself is again a composite object. Within one OMS these versions can be shared among users, in a similar way as with Epochs by selecting the correct version. An advantage of composite objects is that it does not introduce additional mechanisms for exchange. A disadvantage of composite objects is that the contents of a composite object are fully determined by the composition links, and is thus not customisable. This means that all sub-objects of an object are always included in any composite object that includes this object. Furthermore, using composite objects forces a specific organisation upon the user data. A change in the set of objects that must be transmitted often forces a change in the composition structure. Another limitation of composite objects is that there is no flexible mechanism for reconnections. References from the composite object to other objects are static. When the composite object is transported to a different workspace these external references will still point to their original external objects, so these external objects must be available at the receiving workspace.

6.4 Pieces in CAMERA

A piece in CAMERA consists of a set of objects and relationships. To facilitate the discussion in this chapter the term component is used to refer to an object or a relationship. A piece can be extracted from a snapshot and then integrated with another snapshot. This operation is called the piece transplant (see figure 6.1). A piece transplant normally overwrites the contents of the previous version of this piece in the receiving snapshot.

The user has great flexibility in the selection of the contents of a piece. Essentially, arbitrary objects and relationships can be included. Thus the contents of a piece is not a priori restricted to follow certain OMS structures (as is the case with composite objects). Due to the large generality of this approach it can be tailored to arbitrary applications.

All relationships that are part of the piece have to be included explicitly. One slightly different approach we have examined was to automatically include all relationships be-
Figure 6.1: The piece transplant operation

tween objects that are part of the piece. This approach was rejected because it can have unfortunate side-effects. In a hypertext application users can add their own hypertext links between objects. These links would be implemented in CAMERA as relationships. When these objects are transmitted it is not always desirable to include these personal links, because they may not be meaningful to receivers or they may interfere with their private links. Similarly, such relationships that already exist at the receiver should not be deleted by a piece transplant.

Constructing pieces by hand can be done but is very time-consuming and error-prone. This process can of course be automated by writing methods to construct a piece. This approach only works if there is a well-understood algorithm for the construction of the piece. In many cases such an algorithm will fail and interaction with the user is needed in composing a piece. Therefore, CAMERA contains an interactive tool for the creation and modification of pieces, called the piece editor. The piece editor is described in section 6.7.

Objects can occur in multiple pieces at the same time. A piece that represent a program contains program sources, documentation, specification, library files etc. There are several useful subsets that could be represented as pieces in their own right. One piece would contain all objects that are needed by the user of the program, this includes the binaries, library files and documentation, but it excludes the sources. The program itself could use
certain library routines, that are shared with other programs. These library routines could in turn consist of sources, documentation etc. The piece that contained the library routines would be a sub-piece of all programs that use it. When there is a technical writer for the documentation, all the documentation of the program could form one piece, that would be the basis for exchange between the writer and other members of the group. The source program itself is constructed from sub-parts. This structure will be reflected in the internal documentation. In this case a piece can be constructed that contains both sources and internal documentation for sub-parts of the program, which must be updated in parallel. From these examples, it is clear that pieces can contain other pieces. Furthermore, pieces cannot always be organised in a strict hierarchy. In CAMERA pieces are structured in the form of a directed acyclical graph. When one piece contains another piece object, the contents of the second piece are added to those of the first.

CAMERA uses pieces for both repeated exchanges and ad hoc exchanges. At first sight there might seem to be a difference between these two categories. However, in practice the difference is only a gradual one, and only one mechanism seems necessary. In some cases it can be planned in advance that a group of objects will be exchanged on a regular basis. A piece can then be planned carefully. Other exchanges may have a rather ad hoc character. Ad hoc exchanges occur when one user has made modifications that happen to be useful to others, e.g. a bug-fix in a module, a new picture, a new program or a shell-script. Such ad hoc groups can be one-time incidents, that are never re-used again. But experience has learned that often these exchanges will get a regular character. Programs that have been written in a quick and dirty fashion for a one time job, can be (re) used for many years. New developments (e.g. an improved picture, a correction to the bug-fixes) will lead to the exchange of new versions of this group of objects. Therefore, it seems reasonable to store all piece definitions persistently by default. If the user is certain that the piece will never be re-used again it can simply be deleted.

External relationships
After a transplant objects in the piece must be able to establish external relationships with objects in the receiver. There are two different types of external relationships: static and dynamic.

A static external relationship refers to a fixed object, i.e. an objects which object-identifier is known. The relationship that refers to this external object is included in the piece, but not the object itself. After the transplant the relationship refers to the correct object at the receiver. A primary example of this is the instance-of relation. Since it must be possible to determine the class of any object in the OMS, the instance-of relationship for any object in the piece must always be included too. However, it may not be desirable to include the corresponding Class object. When such a fixed external object is not present at the receiver, an exception is raised.

Dynamic external relationships do not refer to a fixed external object, instead the relationship changes dynamically depending on the context of the receiver. When a source program is distributed as a piece, the compiler may be an external object for this piece. It is
not always necessary to use this specific compiler object (otherwise it should be included in the piece), but other compilers may be equally suitable. In this case external relationships must be established dynamically: which object is used depends on the state of the receiving OMS. In the case of a compiler, it might be the case that any C compiler is satisfactory. In this case a relationship with such a compiler can be established dynamically. Dynamic external relationships are handled by using recipes (see section 3.1.4). Recipes are used to compute fragments of a relation. Any recipes in the piece are evaluated in the snapshot after the transplant and can therefore be used to compute the objects to which the external relationship must refer. Because recipes have the full generality of functional methods this is a very powerful mechanism. Individual recipes are represented as objects and can therefore be included in pieces just like other objects. Because of these properties there is no need for a separate mechanism for the specification of dynamic external references.

6.4.1 OMS definitions

Now we will give a more precise description of pieces and their implementation in the OMS. A piece contains a number of objects plus a number of relationships. Pieces are represented in the Snapshot OMS as objects of class Piece.

Class Piece Superclasses Object

Every piece contains a set of objects, that are related to the piece by:

Computed-Relation part-of: Object ↔ Piece

Every piece contains a set of relationships that are described by the computed heterogeneous relation:

Computed-Relation internal-relationships: Piece ↔
System-Relation ↔ **

The first two elements in this relationship describe the piece and relation to which this relationship belongs. The last elements contain the value of the relationship, i.e. the elements in its tuple.

Relationships in the OMS are either computed (by a recipe) or stored. Only stored relationships, that occur in internal-relationships are actually transplanted with the piece. When a computed relationship must be included in a piece, this can be done by including its recipe as an object in the piece. Note that internal-relationships is only used as a mask to determine which relationships are included in the piece, it does not contain the relationships itself. Because internal-relationships is only used as a mask, applications can freely delete relationships without having to update the internal-relationships. Otherwise the piece mechanism would not have been transparent to the applications.

A piece can contain other Piece objects. The components in such a sub-piece are included in the super-piece.

The following relationships are always included with the piece for the transplant:

- The instance-of relationships of all objects in the piece
• The part-of relationships between the piece and its parts
• The internal-relationships for this piece

The instance-of relations are included because the OMS requires all objects to have a class. The part-of and internal-relationships relationships are needed to guarantee that the internal structure of the piece remains intact during a transplant.

6.5 The transplant operation

The transplant operation is used to communicate pieces among environments. The operation consists of two phases: extraction of the piece from its original snapshot, and combining this piece with its target snapshot.

The extraction procedure takes the contents of the piece and stores a representation of the contents of the piece as a new object (of class Album-Piece) in the Album OMS. The class Album-Piece is defined as:

```
Class Album-Piece Superclasses File{
    "Contains an extracted piece as Album object"
}
```

The Album-Piece contains:

• The piece object
• All objects related to the piece object via the part-of relation.
• The instance-of relationships for all objects in the piece
• All stored relationships related to the piece via the internal-relationships relation
• All components from Piece objects in this piece

The extraction operation is defined in the Album OMS as a method:

```
Procedure Extract-Piece (piece:Piece):
    Album-Piece
    "Extract the contents of piece from this snapshot and return it as a new Album-Piece object"
```

The resulting Album-Piece can at this stage be transmitted to other Album OMSes if necessary, using standard CAMERA operations.

Pieces are combined with a snapshot by a similar process. First, it is determined whether the corresponding piece already exists in the receiving snapshot. If this is the case then all objects and relationships that were part of the old piece are omitted from the new snapshot. If the piece did not yet exist nothing needs to be done. Subsequently, the objects and relationships from the Album-Piece are combined with this snapshot. When one of the objects in the piece already exists in the receiving snapshot, the version of this object from the piece will be used instead. This operation is performed by:
Snapshot Procedure Insert-Piece (piece:Album-Piece):
Snapshot
"Insert piece; return the newly created snapshot."

When the piece contains references to external objects that are not present in the receiving snapshot, the transplant operation fails. The user is notified why the transplant fails.

6.6 Example

The example that was introduced in section 5.5.2 will be used to illustrate the definition of pieces. Figure 5.3 is repeated here. Even in this simple example there are many different ways to define a piece that contains \texttt{paper.tex}. One approach is to simply include all components shown in the picture, together in one piece. But some of the include-files, those in directory \texttt{standardMacros}, are standard because they are part of the standard \LaTeX{} distribution. So these could be omitted. Assuming that the receivers already have the \texttt{standardMacros} directory object because it is part of another piece that contains the \LaTeX{} distribution. In this case the relationship (\texttt{paper.tex}, 1, \texttt{standardMacros}) is included in the piece, but the \texttt{standardMacros} object and the entries in the directory are omitted. This situation is shown in figure 6.3.

If it can be assumed that the receiver also has the same locally defined macros, the directory \texttt{localMacros} can be omitted as well. The resulting piece is shown in figure 6.4. This approach would certainly be used if the receiver has private versions of \texttt{psfig.sty}
or `local.sty`. This could happen if the sender is not the maintainer of either of these macros.

Yet another way of defining a piece can be used if `paper.tex` only uses `local.sty` but not `psfig.sty`. This can be detected by examining the depends-on relationships for the derived value (`paper.tex`, `dvi`). In this case the piece could be defined as shown in figure 6.5.

### 6.7 The piece editor

Piece construction cannot be a fully automatic process, because there are too many decisions involved in the selection of a piece. When constructing a piece some information about the potential receivers must be available. When an object in a piece depends on a specific tool, this tool has to be included in the piece if it is not part of the receivers snapshot. However, if the receivers have their own version of the tool, specifically tailored to their environment, it is unwise to include the tool because it overwrites their version at a transplant. Thus in this case, if pieces are to be constructed automatically, information about the receivers must be represented explicitly in the OMS. We concluded that it was too cumbersome to define an automatic procedure for piece construction which was applicable in all foreseeable circumstances; thus attention was focussed on providing tools
which may help in defining pieces explicitly. **CAMERA** therefore contains such a tool called the *piece editor*. The piece editor helps the user to examine and modify the contents of the piece.

The piece editor has a rule-base with heuristics that describes which objects and relationships normally form part of a piece. The user can decide to include or exclude objects and relationships in/from a piece. The piece editor uses its rule-base to suggest which additional object/relationships could also be included. The user can always override these suggestions. The rule engine itself does not modify the OMS. Only when the user has agreed with the suggestion, the objects and relationships are added to the piece by updating the `part-of` and `internal-relationships` relations.

The rule-base contains knowledge about the structure of the OMS and uses this to generate its suggestions. For example, the rules could describe that when a C source file object is included in the piece, all include-files must be added too, except for the standard include files, that are found in the **CAMERA** equivalent of the `/usr/include` directory. Because the rule-base can be changed, it can also describe different behaviour, e.g. to omit the include-files from `/usr/local/include`, or to include all include-file regardless of their location. Different types of pieces, i.e. different sub-classes of `Piece` can use different rule-bases, and can therefore behave differently.

The `depends-on` relation (described in section 5.5.1) is important input for the piece editor. This relation contains all inputs that were accessed during the computation of a derived value. If a derived value is part of a piece, then this derived value can only be reproduced by the receiver when the inputs are available. This means that they must be included in the piece, unless it is expected that these inputs (or equivalents thereof) are already present at the receiving side. The rule-base can be programmed to include all inputs automatically. For many simple functional methods that produce derived values this is exactly the desired behaviour, thus such methods can be added to the OMS without requiring any modification to the rule-base at all.

Because the piece editor is run as a separate tool, it is not a real primitive of the **CAMERA** system. When users are not satisfied with the piece editor they can replace it with their own...
tool. Pieces themselves are a primitive in Camera, but they appear to be sufficiently flexible. They can be used to mimic other approaches such as composite objects.

6.7.1 Using the piece editor

The user can interact in several different ways with the piece editor.

- Including and excluding objects/relationships
- Setting switches (a kind of options)
- Interacting with the rule engine.

Including/excluding objects

For the construction of most pieces the user is not directly confronted with the rule engine. The user only selects objects/relationships that must added to or deleted from the piece. The piece editor then adds/deletes further components as a consequence of these additions/deletions. Components can be selected by specifying a query to the OMS. A query can be an arbitrary message that returns one or more components.

Switches

A second level of interactions is by setting switches. Switches are options that influence the behaviour of the rule engine. An example of such a switch is the decision to include standard libraries in the piece or not. Switches can be read and set. Switches are local to a piece, and different types of pieces can have different switches associated with them. Therefore, users are not confronted with all possible switches that exist in the OMS but only with those that are relevant to the current piece.

Switches exist mainly for convenience purposes. They allow a simple way to modify the behaviour of the rule engine without being confronted with its internals.

The value of a switch for a piece is recorded in the relation:

\[
\text{Computed-Relation} \quad \text{piece-switch: Piece } \leftrightarrow \text{ string } \leftrightarrow \text{ string}
\]

"Record the value of switch for a piece. The second element is the name of the switch. The third element is the current value of the switch."

The rule engine examines this relation to determine the values of the switches.

Interaction with the rule engine

In complicated cases users may want to examine or change the rules. Rules are partitioned in rule-sets, and each piece has a number of rule-sets associated with it. Each rule-set is stored in a different object. This makes it easier to exchange them. Each sub-class of the class Piece can have an associate rule-set. Each sub-class will "inherit" the rule-sets from its ancestors. Different sub-classes can have different rule-sets and can therefore behave differently with respect to piece construction. Additionally, there is one local rule-set, that is private to the piece. It contains the rules that are unique to this piece. In most cases the
local rule-set is empty, because the standard rule-sets are sufficient to describe the piece. Because rule-sets are shared among pieces, modifications to such rule-sets are re-used by other pieces as well.

Piece construction is not a simple process, and rules can get complicated. The piece editor can explain why it made a certain decision by showing the relevant rules. It may also give an overview of all applicable rules for a given component.

### 6.7.2 Modifying a piece

The actual contents of a piece can change in two ways: by using the piece editor and fully automatically.

A piece can be adapted automatically by using recipes and/or triggers. Both the part-of relation and the internal-relationships relation are computed relations, and this can be used to change the contents of a piece dynamically without user interaction. Another mechanism that can be used to provide automatic adaptation of a piece is by using triggers. The guard states the conditions under which a piece should be modified and the action part of the trigger performs the modifying operations. The most logical way to add recipes and triggers is during creation of the piece. Different kinds of pieces require different recipes/triggers. This is handled by creating different sub-classes of Piece that are tailored to handling different objects in the piece.

Fully automated piece modification by recipes and triggers should only be used if the rules for adding objects and relationships are well understood, in the sense that they can always run without user intervention. They will be used when an object always requires specific accompanying objects to be included in the piece. In cases where this process cannot be fully automated, such dependencies between objects should be encoded as rules for the piece editor. The user can then check the resulting piece, and optionally override decisions made by the rule engine.

### 6.7.3 Organisation of standard components

Some objects must not be transmitted because they are to be considered as standard objects, of which a version will be available at the receiver. Examples of such objects are class objects and tools such as compilers. In such a case including these objects in the piece may be both expensive, due to possibly higher transportation cost, and harmful, because they overwrite the tools at the receiver. The recognition of standard objects is important for the formation of pieces, but some organisation is needed to accomplish this. In the OMS it may be difficult to distinguish standard objects from non-standard ones. An example of a standard object is a standard include-file for the C compiler (normally found in /usr/include), which from the systems point of view is very similar to include-files created by the user. In such a case the distinction can only be made based on the directory in which it is located.

There are several policies to the recognition of standard objects. The following policies are among the possible:
• Put standard objects in standard directories, analogous to /usr/bin and /usr/lib. A disadvantage of this approach is that it cannot easily be extended to recognise standard relationships.

• Add a special attribute or special relationship to all standard objects. This essentially has the same problems as the previous one.

• Add rules to the rule-base for identification of standard objects, e.g. a list of standard objects that are known to the piece. This is a very powerful approach, that can easily identify both objects and relationships. However, the rule-base should be organised so that it is not necessary to add rules for each new object.

• Put standard objects in pieces. It is very likely that in CAMERA standard objects are part of a piece anyhow because it makes it easier to exchange these objects, e.g. for the distribution of new versions. This property can now also used to recognise these objects as standard ones. It can be used to describe both standard objects and relationships.

The rule-base that is described in the next section is sufficiently flexible so that each of these policies can easily be expressed. Which of these policies is chosen, is up to the user.

The default rule-set in the current prototype is based on the last policy, where standard objects are recognised by the fact that they already form part of a piece. The default policy for piece construction is to exclude objects/relationships from the piece that already are part of another piece, unless of course the latter piece is part of the former. Because this policy is essentially tool independent, the addition of new tools will only very rarely require modifications to the rule-base.

6.8 The rules

The design of a good rule mechanism to describe the construction of pieces is far from trivial. This section lists requirements and the next section describes the current version of the rule mechanism.

The rules must be able to describe both inclusion and exclusion of components. For example, a rule must be able to describe that all inputs for a derived value in the piece must also be included. Similarly, it must be possible to describe that all components that form part of another piece must be excluded.

We require that the result of evaluating a set of rules is independent from the order in which the rules occur. Any dependency on the ordering will hinder re-use of the rule-base, because a small change in the ordering will influence the final outcome. To illustrate this problem, assume that one rule adds component A, the second rule adds B when A is present, and a third rule removes A. If the outcome of the rule evaluation is order-dependent these example rules give four possible outcomes by using different permutations of the rules. Because rules are collected in different rule-sets the ordering of the rules will depend on the ordering of the rule-sets. This makes it difficult to specify general rules, that are
always satisfied in the constructed piece. For example, assume there is a general rule in one rule-set to include B when A is present, and another rule in a second rule-set that include A. Now B is only included if the second rule-set is evaluated before the first one. Thus general rules must always be part of the last rule-set that is to be evaluated. Because rule-sets can be exchanged among different users and can be combined, a fixed evaluation order among rule-sets cannot be guaranteed. Including all general rules in all rule-sets is the only way to guarantee that they will always be satisfied. This is highly unsatisfactory and causes large maintenance problems when the rules must be adapted. Due to these problems, an order-independent rule mechanism is to be preferred.

It must be possible to specialise rules, without having to modify the rules themselves. For example, a general rule could have been defined that adds all inputs for derived values whose objects have been included in the piece. However, there could be exceptions to this rule, e.g. if standard tool objects, such as compilers, should not be included in the piece. These exceptions could be handled by modifying the original rule, but then rules with many exceptions would grow complicated. Furthermore, rules have to handle all specialisations at once, which makes them less re-usable.

The rule engine should not be able to modify the OMS during rule evaluation because this may influence the outcome of the evaluation of the other rules.

### 6.8.1 The rule mechanism

The rules have a Prolog-like appearance. They can contain meta-variables that stand for components. The rule engine attempts to find an assignment of components to these meta-variables that satisfies the rules. There are two different types of rules that include or exclude components. All rules have a priority associated with them. Matching rules with a higher priority override those with a lower priority. This is used to specialise or undo the effect of rules in other rule-sets without having to modify them. Having a fixed priority for the rules can be problematic when rule-sets are exchanged, because a rule-set depends on certain assumptions about the priorities that are used in other rule-sets. This problem can only be overcome by using conventions about the priorities that are used. The number of different priority levels that are needed is not very large. Furthermore, the most important rule-sets are pre-defined in CAMERA, and most new rule-sets only extend these standard rule-sets.

The following syntax is used for describing the rules:

```
include(Comp,priority) if condition
exclude(Comp,priority) if condition
```

Here Comp is a meta-variable that stands for the component, and priority is an integer that gives the priority for this rule. condition contains the conditions that must be satisfied.

There are two different kinds of components, objects and relationships. Relationships are notated as [relation,value1,value2,...]. Here relation is the name of the
relation to which this relationship belongs, and value1,... is a list that represents the value of the tuple.

Each condition consists of a list of predicates. The component is included if all predicates can be satisfied. Predicates have arguments that are either constants or meta-variables. Predicates can also be negated by using not. The rule engine attempts to find an assignment for values of the meta-variables that satisfy the predicates. If an assignment is found that satisfies all predicates in the condition of a rule, the condition is satisfied, and the component is included/excluded with the given priority. Whether the component is actually included in the piece, depends on the effect of rules with a higher priority that match this component.

The following predicates and constants are available.

\[ \text{relation-name, field1, field2...} \]

is true if there is a tuple \( \{field1, field2...\} \) in relation relation-name.

class-of(Object, Class-name)

is true when Object is an instance of Class-name, or of one of its sub-classes.

query(Relation, Query, Result)

is true when Result is one of the tuples in the result of executing Query on Relation.

current-piece

is the object-identifier of the piece that is currently constructed.

in-piece(Comp)

This predicate is true when the component is currently included in the piece by the piece editor.

When a relationship is included frequently the objects to which this relationship refers must be included, too. For this purpose special rules (include-all and exclude-all) are included as a form of shorthand. They include/exclude both the matching tuple and all objects in this tuple.

\[ \text{include-all([directory,X,*],Z),100} \text{ if in-piece(X)}. \]

This rule includes both the relationship \([directory,X,*],Z\) and the object Z.

The following is an example of the rules:

\[ \text{include([instance-of,X,Y],1000) if in-piece(X)}. \]

This rule includes all instance-of relationships for objects that are included in the piece.

The in-piece predicate is needed because rules must be able to determine whether a given object is already included in the piece. However, it complicates the evaluation of the rules because the inclusion of a component can cause several other rules to match.

It is possible to construct rules that conflict. The rule engine detects this and gives a warning.
6.8.2 Examples
We now give some examples of the usage of the rule mechanism. In these examples the convention is used that the normal rules have priorities < 1000, important rules that should normally not be overridden by the user have priorities $\geq 1000$.

Example 1:
The following rule automatically adds all instance-of relationships for objects that are included in the piece. This rule can be considered to be built-in, because the OMS has to be able to determine the class for all objects.

\[
\text{include([instance-of,X,Y],1000) if in-piece(X)}.
\]

Example 2:
The next two rules include part-of relationships for all objects in the piece. Furthermore, the first rule also includes all sub-objects of pieces that have been included in the current piece. The second rule includes all relationships that are part of a sub-piece.

\[
\begin{align*}
&\text{include-all([part-of,X,Y],1000) if in-piece(Y).} \\
&\text{include(X,1000) if in-piece(Y),} \\
&\quad \text{[internal-relationships,Y,X].}
\end{align*}
\]

Example 3:
When classes are transplanted, their super-class relationships must also be included. This is achieved by:

\[
\text{include([super-class,X,Y],1000) if in-piece(Y)}.
\]

Example 4:
When a class is transplanted, it is normally desirable to include all its methods. This rule has a lower priority than the previous one, so that it can be overridden by other rule-sets.

\[
\begin{align*}
&\text{include-all([method-for,X,*,Y],100) if in-piece(Y).} \\
&\text{include(X,0) if in-piece(Y),} \\
&\quad \text{[depends-on,X,*,*,derived,Y,*,*].}
\end{align*}
\]

Example 5:
Frequently, all objects on which a given object depends should be added. Note that this rule works for arbitrary derived values. Thus for new tools it is not needed to add new rules that describe their inputs.

\[
\begin{align*}
&\text{include(X,0) if in-piece(Y),} \\
&\quad \text{[depends-on,X,*,*,derived,Y,*,*].} \\
&\text{include(X,0) if in-piece(Y),} \\
&\quad \text{[depends-on,R,query,Q,derived,Y,*,*],} \\
&\quad \text{query(R,Q,X).}
\end{align*}
\]

Example 6:
Structural information about the objects, i.e. the relations they have can also be used for piece construction. Assume that there is a documents relation that relates the documentation for some object to this object. The example rule includes the documentation object for each object in the piece. The inclusion of the documentation object may in turn force the inclusion of other objects it needs, e.g. the previous rule will automatically include all inputs for this documentation object.

\[
\text{include-all([documents,X,Y],1) if in-piece(Y).}
\]
Example 7:
The next example shows the usage of switches to modify the behaviour of the rule engine. This rule adds all sub-directories of objects that are in the piece. The user can simply modify a switch without having to be aware of the underlying rules.
\[
\text{include-all([directory,X,*,Z],100) if}
\]
\[
\text{in-piece(X),}
\]
\[
\text{[piece-switch, current-piece, add-sub-directories, Yes].}
\]

Example 8:
Section 6.7.3 described several policies for recognising and excluding objects because they are "standard". The following rule implements the policy to recognise standard objects by the fact that they are part of a piece that is an instance of the class StandardPiece (a sub-class of Piece). This rule automatically excludes such objects.
\[
\text{exclude(X,100) if part-of(X,Y),}
\]
\[
\text{class-of(Y,StandardPiece),}
\]
\[
\text{not in-piece(Y),}
\]
\[
\text{not [part-of, current-piece, Y].}
\]

The rules above are sufficient to handle the piece construction example from section 6.6. It is assumed that the standardMacros directory plus its contents are part of a StandardPiece that contains the standard \LaTeX{} distribution. Then rule 8 is sufficient to exclude all components that are part of the \LaTeX{} piece from a newly constructed piece.

When the contents of the localMacros directory have been added to a StandardPiece that contains the local \LaTeX{} extensions, these are also excluded from the piece. Thus we obtain the piece shown in figure 6.4.

When the contents of the localMacros directory have not been added to a StandardPiece they can be included in the piece. Rule 5 automatically adds all inputs that were used to produce by paper.tex. Thus if it only needs local.sty the constructed piece is as shown in figure 6.5. If it also uses psfig.sty the result is as shown in figure 6.3.

The example pieces from figure 6.3 and 6.4 can also be constructed without using the depends-on relation. The following rule constructs the piece from figure 6.4.
\[
\text{include([TeXMacroDir,X,*,Y],100) if in-piece(X).}
\]

Assuming that there is a StandardPiece for the \LaTeX{} distribution, the following rule together with the previous rules construct the piece from figure 6.3.
\[
\text{include-all([directory,X,*,Z],0) if}
\]
\[
\text{in-piece([TeXMacroDir,Z,*,Y])}.\]

Note that the last two examples that do not use the depends-on relation, are very application dependent and include components that were not needed.
6.9 Discussion

Interfaces

One possibility that was examined during the design of pieces was to provide each piece with an interface that describes its interactions with the outer world. This interface would then describe the dependencies of this piece on services outside this piece. Similarly, the interface would also describe the services that were offered by the piece. A similar idea is found in modularised programming languages (e.g. Ada [Ada82] or Modula-2 [Wir83]) where each module has an interface that describes its relations with other modules. For example, in CAMERA a piece that contains a program plus its sources would require services like the compiler and standard libraries, and it would provide the functionality of the compiled program to the rest of the OMS.

This idea was rejected because it had several important disadvantages. It takes extra time to declare interfaces for new pieces. Frequently interfaces will simply list all objects in piece, thus adding no extra information. There is also the additional maintenance problem here: when a piece is updated its interface must be updated.

A functional specification of the rules

An alternative mechanism for the piece editor would have been to use a functional program to construct the pieces. Rules would be implemented as functions, that take a piece as input and return a modified value of the piece. The piece would be constructed by applying all rules to a piece that is initially empty. Rules with a lower priority would be applied before rules with a higher priority. In this way high priority rules can override the decisions made by lower priority rules.

This approach was rejected because assigning priorities to the rules is very difficult and can only be done if the rule designer is aware of all interactions among the rules. Suppose the rules had to be designed in such a way that component A is included in the piece if and only if component B is present in the piece. This rule can only be enforced if it is executed (i.e. has a higher priority) after all rules that might influence the inclusion or exclusion of component B. Thus the rule designer must know which rules might add or remove a specific component. If the language that is used to describe the programs for the rules is sufficiently powerful, this problem is undecidable. The interactions among rules also make it difficult to combine rule-sets, especially if some of these were written by others.

The rule mechanism that was chosen for the piece editor does not impose an ordering on the execution of rules. With this mechanism it is simple to describe a rule that included A if B is present in the piece. The priority of this rule is completely independent from those of the rules that determine the inclusion of A. Therefore, in this approach it is much easier to write rules and combine them.
6.10 Conclusions

The piece mechanism offers a flexible approach to grouping components for exchange. Due to the great flexibility of the mechanism it is independent from the application that uses these components.

In many traditional systems there is little support for collecting components. It occurs frequently that objects are forgotten when a set of objects is transmitted. The CAMERA piece editor helps the user in constructing pieces. It has a powerful rule-base that contains rules that describe the formation of pieces. The rule mechanism automates the construction of pieces to a large extent. The user however can easily override or modify the suggestions of the piece editor.

The combination of derivation management and the rule-base of the piece editor is a powerful one. All inputs for a derived value in the piece must either be part of the piece itself or be available at the receiver. The CAMERA dependency management automatically traces all inputs for derived values and this information is then used by the rule engine.
7 Merging

7.1 Introduction

Parallel activities, where each user has private copies of shared data occur frequently in many different kinds of development situations, and this is the normal way of working in CAMERA where each user has a completely private workspace in the form of an environment. After a certain period the results of these separate lines of development must be merged. Merging is a difficult process, certainly when many interrelated objects are involved. Tool support for this merging process is desirable, but hardly existent.

We first give a short overview of existing tools for merging in section 7.2. Existing merge techniques are state-based: they only use the initial and final state of a development. This chapter describes a different strategy called operation-based merging that uses the operations that were performed during the developments. Operation-based merging works for arbitrary abstract data-types, and guarantees that data-type invariants are respected. Operation-based merging is introduced in section 7.4. Later sections describe algorithms for detecting and solving conflicts based on this approach.

7.2 Existing merge tools

Several systems are available that support some form of merging. These systems mainly support merging of individual text files and are described in section 7.2.1. Merge tools providing support for merging other kinds of components, are dealt with in section 7.2.2. Unfortunately most existing merge tools are not very satisfactory. A list of deficiencies is identified in section 7.2.3.

7.2.1 Text-based merge tools

Several systems support merging of ASCII text files, e.g. RCS [Tic85], Sun’s filemerge-tool [AGMT86], and DSEE [LC84]. All these systems are line-based, and attempt to detect common lines and lines that have been inserted/deleted or moved. They do so by finding the longest common subsequences between the texts to be merged.

1 A variant of this chapter will be published as [LvO92].
These tools either use two-way merging, which attempts to merge the two final versions, or three-way merging which also uses the common ancestor of the versions to be merged. Three-way merging is more powerful than two-way merging because more information is available. Three-way merges are able to detect deletions, while two-way merges obviously cannot detect these. If a line is only available in one of the versions, some two-way merge algorithms assume that this line has been added and by default include it in the finally merged result. This however is not always the desired behaviour. When the line has been deleted in one of the final versions (but not in the other) the correct default behaviour is to delete the line in the merged version. Furthermore, if a certain line of a file has different values in both versions, two-way merges cannot make a decision which one to choose. three-way merge can detect whether one of the values is equal to the value of the line in the original file, and use the other value by default in the merged version.

In Tandem's version control system [SK90] individual text lines are tagged with a unique identification and recorded in a data-base. Modifications create new lines with different identifications. These tags are used to detect insertions/deletions, and this information is then be used to perform merges. The algorithm currently does not seem to handle the operation of moving lines to a different position.

7.2.2 Other merge tools

Some tools are available that do not work on individual text files.

The combination of the Unix diff program and Larry Wall's patch [Wal88] can be used to merge sets of text files. diff is used to generate context diffs that list both the original and the changed lines plus a small number of surrounding context lines. patch can then be used to apply the diffs to another file hierarchy. patch has a certain amount of flexibility. The files to which it is applied need not be identical to the versions that were used to compute the diff. Differences occur if the receiver has already modified the original file. When patch cannot find the original lines at the location it expects, it examines the neighbourhood of this location. If matching lines are found, it applies the delta at this modified location. This approach only works if the lines at this location have not been modified themselves. Although patch is useful in many practical circumstances, it is still rather limited. When deltas cannot be applied they must be handled manually.

Horwitz and Reps [HPR88] have developed a method for merging programs in a very simple language that is based on the semantics of this language. This approach is very interesting but unfortunately only works for their very limited language. It is hard to extend the approach to "real" programming languages.

Sun's NSE [CFH88] provides merging of environments. An environment contains a set of files plus some specialised objects. During the merging process NSE compares the contents of the environments and it invokes a file merge tool for merging individual text files. NSE also contains small data-bases, essentially no more than a list of (key,value) pairs. It also provides tools to merge these.
Westfechtel [Wes91] has developed a method for merging trees. This can be used to merge programs (represented as parse trees). This method takes some contextual dependencies into account. It tracks the declaration and use of identifiers. Thus it is able to handle the renaming of identifiers in one line of development.

The PACT merge tool [PAC89] provides support for merging composite objects in PCTE. It is used to merge composite objects that have a similar structure, described by the same merging template. It only merges structural links, and cannot be used to merge the contents of the atomic objects that form the composite object.

7.2.3 Deficiencies

The main deficiencies of these systems can be summed up as follows:

- Text based tools suffer from the following problems:
  - They cannot handle changes within a single line. Thus if a line has been modified in both lines of development, only one of the versions can be selected. Thus the user has to integrate the other version manually. Since some tools do not support edit operations while merging, this requires an explicit separate bookkeeping by the users, which is a cumbersome and error-prone process.
  - There are several operations that change the contents of many lines, without having a real semantic meaning. Examples are the pretty-printing of a program and reformatting text to give all lines a similar length. In this case merging tools flag many lines as changed and detect many conflicts.
  - A similar problem exists when in a program the indentation is changed, e.g. when an if statement is added around an existing sequence of statements. The indentation of all lines is changed, and therefore all these lines are regarded as changed during the merge.

- The merging can only be applied to a limited set of types. Most merge tools are limited to merging texts. Furthermore, merge tools for merging structured sets of objects are hardly available. The PACT merge tool handles the relationships between arbitrary types of objects but cannot merge the contents of any object type.

- These systems do not offer much help when conflicts are detected. In most cases when a conflict arises the user must choose a value from one of the lines of development and if necessary perform manual editing afterwards.

- They are value based in the sense that they use no information about actual developments, but only information about the initial and final states. This can lead to unnecessary conflicts, conflicts that go unnoticed and to incorrect results. An example of the first is the case where in both lines of development the resulting source is pretty-printed. Even when the "real" modifications do not conflict, pretty-printing leads to conflicts. An example of the second type is when one user changes the name of a procedure while the other adds a new call to this procedure. After merging there is one call with the old procedure name. An example of the third is
a counter for the total time that has been spent by programmers on the project. In each line of development will regularly invoke an operation to increment the local value of the counter with the time that has been spent in this line of development. State-based merges will not detect a conflict when an equal amount of time that has been spent in each line of development, but will simply use the resulting value of the counter. When the amount of time differs, state-based merges will offer the user the selection between one of the two final values, but not the option to add the sum of the increments, which would be the correct answer.

- They offer hardly any support for merging modifications that change the structure among objects. For example, none of the file-based merge systems can handle file-renaming in a decent fashion.

### 7.3 Design goals

The design goals for a merge procedure for the CAMERA system were the following:

- Merging must work on entire snapshots.
- CAMERA should give flexible support for merging different object types, thus not be restricted to specific classes of objects. It must also be possible to merge instances of user defined classes.
- As many conflicts as possible should be detected, and, when possible, these should be resolved automatically. Extensive support is more important than execution speed. Merging is a difficult process where every bit of support is helpful. Furthermore, merging does not occur very frequently.
- The merge tool must have a good user interface, in order to present conflicts and show potential solutions. Furthermore, the number of decisions by the user must be minimised. Completely automatic merging is not possible due to the only partially known semantics of objects and changes. Therefore, user intervention cannot always be avoided.

### 7.4 Operation-based merging

All existing merge tools are value based. The operations that have been performed in each line of development are ignored, and only the end-results count. The previous sections have shown some of the problems with this approach.

We therefore propose a different approach that is based on the operations that were performed. In CAMERA these operations are recorded in the form of transformations. Transformations are composed of sequences of OMS-operations that are called primitive transformations. Transformations are treated as functions, that take snapshots as inputs and return new snapshots as result. Thus $T_n(s)$ is the snapshot that is the result of a line
of development whose operations are recorded in $T_a$ when the starting point was the
snapshot $s$.

**Basic Assumption for operation-based merging:** Good candidates for the
merged result of transformations $T_a$ and $T_b$ on an initial snapshot $s$ are $T_a(T_b(s))$ or $T_b(T_a(s))$.

When we are lucky the results of applying the transformations in different orders are
equivalent. In this case this result is chosen as the merged result from both transformations.
We could have decided to use equality instead of an arbitrary equivalence relation to
compare values. However, using an equivalence relation gives some added flexibility, that
allows ignoring unimportant differences between values.

Given an equivalence relation (i.e. a reflexive, symmetric, and transitive relation) $\approx$, we
define two operations $T_a$ and $T_b$ to **commute locally** on an input $s$ iff $T_a(T_b(s)) \approx T_b(T_a(s))$. In a similar way we define that two operations $T_a$ and $T_b$ **commute globally**
iff $\forall z : T_a(T_b(z)) \approx T_b(T_a(z))$.

When two transformations commute locally on their initial snapshot, the final result is a
good candidate for the result of the merge.

**Detecting conflicts**

In many important simple cases the two transformations commute locally. For example,
if $T_a$ and $T_b$ operate on disjoint sets of objects, operation-based merging works.

However, in many other cases the transformations do not commute locally. In these cases
we cannot merge automatically and some form of user intervention is still necessary. One
possibility is to allow the user to edit the two transformations until they do commute. But
it is possible to be more helpful, by examining the primitive transformations from which
a transformation is made. In general, it can be expected that only a very small subset of
these primitive transformations actually conflicts. These groups of conflicting primitive
transformations are isolated and presented to the user, together with possible solutions.
The details of this process are described in section 7.5.

**Solving conflicts**

When a set of conflicting primitive transformations has been detected, these conflicts must
be solved by the user. The merge tool illustrates the conflict by showing the initial state of
the OMS, the two sets of conflicting primitive transformations (one set for each of the two
original transformations), and the two states that result if these sets are applied in different
orders.

The user can select one of the following approaches:

- Resolve conflicts by **imposing an ordering** on the primitive transformations. For
  example, if a pretty-printer is invoked on a source file in one development line
  while small changes have been made to the file in the other line, many conflicts can
be avoided by first applying the changes and then pretty-printing. Similarly, if an identifier is changed in one line of development (using a global substitute command) and a new use of the old identifier is added in the other line of development, the conflict can be solved by performing the substitute command last.

- Resolve conflicts by deleting a primitive transformation. When multiple calls to a pretty-printer are a source of conflicts, normally all but the last one can be deleted without harm.
- Resolve conflicts by editing the transformation. If the previous options are not sufficient it is also possible to modify the transformations by modifying their primitive transformations or adding new ones.

The last two solutions may change the existing conflicts. The merge tool needs to check for this situation.

7.4.1 Advantages of using operation-based merging

Operation-based merging is very attractive for the following reasons:

- The approach is extensible, it can be applied to all object types (even user-defined types).
- It can be used to merge entire object systems.
- It avoids many of the conflicts mentioned above.
- It can detect conflicts that are not noted by value based approaches.
- The merged result is more likely to be semantically consistent than with the state-based approach. Normally, operations respect invariants of the object. A data-base normally stores its contents in an internal binary format. It is impossible to merge the contents of the data-base unless the internal binary format is known to the merge tool. However, if the data-base access calls are used in the merge process, then the internal structure is respected.
- Operation-based merging is more general than value based merging. State-based merging can be seen as a very restricted form of operation-based merging, that attempts to reconstruct transformations after the fact. This is difficult, since there usually is more than one function, that can be used to transform an initial state into the final state. With operation-based merging the operation that is actually used is known.

7.5 Overview of the merge process

This section outlines how the merging process for merging transformations works. Later sections describe some aspects in a more detailed fashion.
Step 1: Forming raw blocks

Every transformation consists of a sequence of elementary steps called primitive transformations. E.g. $T_a X = T_{a,n} T_{a,n-1} \ldots T_{a,1} X$. These are the smallest grain of operations that have been recorded.

Operation-based merging attempts to find sets of conflicting primitive transformations, called blocks. The transformations in a block may conflict with one another, while transformations from different blocks do not conflict.

The global structure for the merging process is as follows:

1. The first step uses global commutation information to form raw blocks of primitive transformations, using a transitive closure algorithm (see section 7.6).

2. The second step takes each raw block in turn and uses local commutation information to detect conflicting operations.

3. The conflicts are then presented to and solved by the user. The resulting transformation is then used to create the merged result.

7.6 Step 1: Forming raw blocks

The first step only uses information about the global commutation behaviour of a set of operations, and groups the primitive transformations of the transformations that are to be merged into raw blocks.

These blocks are constructed by imposing a (pre-)ordering (the "before" relation) on the primitive transformations. The before relation indicates whether a given primitive transformation must be performed before the other. Cycles in the ordering determine the raw blocks. Blocks form a partitioning of the union of the primitive transformations in such a way that each block is the smallest set such that each cycle is contained within one block. Different blocks can be ordered with respect to one another, but obviously there cannot be any cycles among blocks.

The inputs are formed by two transformations plus a procedure to determine whether two primitive transformations commute globally. The procedure attempts to minimise the number of times that this procedure is called, because this could be an expensive operation. Some methods to determine whether two operations commute globally are described in section 7.8.

For the description of the algorithm we use the following notation. The set of primitive transformations is denoted by $T$. The two transformations that are merged have length $L_1$ and $L_2$, respectively. The primitive transformations are recorded in $\tau$. It is assumed that there is a way to determine whether or not two transformations commute globally. The result is stored in the relation conflict, thus $a$ conflict $b$ is true if $a$ and $b$ do not commute globally. The predicate $a$ before $b$ is true if primitive transformation $a$ must be performed before $b$. The before relation must be transitive. If two primitive transformations within the same transformation conflict, the one with the lowest index must be performed before the other. Cycles in the before relation (i.e. $a$ before $b \land b$ before $a$) indicate merge conflicts,
because \(a\) and \(b\) cannot be ordered and a decision must be made by the user. Conflicts between primitive transformations \(a\) and \(b\), that are part of different transformations always lead to a cycle.

The data-structure and procedure are described here in a Z-like syntax (see [Spi89]).

\[
\begin{aligned}
L_1, L_2 : \mathbb{N} \\
\text{TrIndex} \equiv \{1\} \times 1..L_1 \cup \{2\} \times 1..L_2 \\
\tau : \text{TrIndex} \rightarrow T \\
\text{conflict} : T \rightarrow T \\
\text{before} : \text{TrIndex} \rightarrow \text{TrIndex}
\end{aligned}
\]

Three different algorithms for computing raw blocks are described and compared. The basic algorithm compares all primitive transformations (\(#\text{TrIndex}(\#\text{TrIndex} - 1)/2\) pairs) and then computes the transitive closure.

\[
\textbf{Algorithm 0}
\]

\[
\text{before} := \\
\{i, j : \text{TrIndex} | \tau(i) \text{conflict} \tau(j) \land (\pi_1 i \neq \pi_1 j \lor \pi_2 i < \pi_2 j)\}^+
\]

A disadvantage of Algorithm 0 is that it determines for all pairs of primitive transformations, whether or not there is a conflict. This could be an expensive operation and it seems sensible to reduce the number of comparisons as much as possible. This can be done by using the transitivity properties of \(\text{before}\). When it is known that \(a\ before\ b\ \land\ b\ before\ c\) it can be deduced that \(a\ before\ c\). Thus in this case there is no need to compare \(a\) and \(c\) in order to know whether \(a\ before\ c\) holds. The following two algorithms use this transitivity property to reduce the number of comparisons.

They further attempt to minimise the number of comparisons by selecting the best candidates for these comparisons.

Algorithm 1 uses a function \(\text{new}(\text{before}, a, b)\) that gives an upper bound on the number of entries that would be added to the \(\text{before}\) matrix if primitive transformations \(\tau(a)\) and \(\tau(b)\) don’t commute.

The algorithm maintains a set \(\text{uncomputed}\) of pairs of indices of the primitive transformations in both transformations that have not yet been compared. From this set it selects the pair \((i, j)\) that would add the highest number of entries to the \(\text{before}\) matrix if it could be determined that the corresponding primitive transformations conflicted. This pair \((i, j)\) is removed from \(\text{uncomputed}\). If the corresponding primitive transformations could add new entries to the \(\text{before}\) matrix it is computed whether they actually conflict. If this is the case the \(\text{before}\) matrix is updated.
Algorithm 1

uncomputed := \{u, v : TrIndex | u < v\}
before := \\emptyset
while uncomputed \neq \\emptyset do
(i, j) \in uncomputed \bullet
\text{new}_1(\text{before}, i, j) =
\max_{(i', j') \in \text{uncomputed}} \text{new}_1(\text{before}, i', j')
uncomputed := uncomputed \setminus \{(i, j)\}
if \text{new}_1(\text{before}, i, j) > 0 then
\text{if } \tau(i) \text{ conflict } \tau(j) \text{ then}
before := before \cup \text{add}(\text{before}, i, j)

\text{S}(\text{before}, i) = \{t : \text{TrIndex} \mid t \text{ before } i\} \cup \{i\}
\text{L}(\text{before}, i) = \{t : \text{TrIndex} \mid i \text{ before } t\} \cup \{i\}

\text{add}(\text{before}, i, j)
= \text{S}(\text{before}, i) \times \text{L}(\text{before}, j) \cup \text{S}(\text{before}, j) \times \text{L}(\text{before}, i)
\text{when } \pi_1 i \neq \pi_1 j
= \text{S}(\text{before}, i) \times \text{L}(\text{before}, j)
\text{when } \pi_1 i = \pi_1 j \land \pi_2 i < \pi_2 j
= \text{S}(\text{before}, j) \times \text{L}(\text{before}, i)
\text{when } \pi_1 i = \pi_1 j \land \pi_2 i > \pi_2 j

\text{new}_1(\text{before}, i, j)
= \#(\text{add}(\text{before}, i, j))

The function \text{new}_1 is not very precise, in that it only provides an upper-bound on the number of entries that are added to the before matrix. A variant uses the actual number of entries that are added. The only essential difference is in the function \text{new}_2 which omits existing entries in before from the count.
Algorithm 2

\[
\text{uncomputed} := \{u, v : \text{TrInd}x | u < v\}
\]
\[
\text{before} := \emptyset
\]
while uncomputed \(\neq \emptyset\) do
\[
(i, j) \in \text{uncomputed} \bullet \text{new}_2(\text{before}, i, j) = \max_{(i', j') \in \text{uncomputed}} \text{new}_2(\text{before}, i', j')
\]
\[
\text{uncomputed} := \text{uncomputed} \setminus \{(i, j)\}
\]
if new$_2$(before, i, j) > 0 then
\[
\text{if } \tau(i) \text{ conflict } \tau(j) \text{ then}
\]
\[
\text{before} := \text{before} \cup \text{add}(\text{before}, i, j)
\]
\[
\text{new}_2(\text{before}, i, j) = \#(\text{add}(\text{before}, i, j) \setminus \text{before})
\]

Notes:
The function \(\text{new}_2(\text{before}, i, j)\) can be implemented as an array. When a new tuple \((i, j)\) is added to \(\text{before}\), at most those elements \(\text{new}_2(\text{before}, u, v)\) must be updated for which \(\{i, j\} \cap (S(\text{before}, u) \cup S(\text{before}, v) \cup L(\text{before}, u) \cup L(\text{before}, v)) \neq \emptyset\). Furthermore, when \(\text{new}_2(\text{before}, u, v) = 0\) or when \((u, v) \in \text{uncomputed}\) there is also no need to re-compute \(\text{new}_2\).

The selection of the \(i, j\) with the highest value of \(\text{new}\) can be implemented by storing all values of \(\text{new}_2\) in an appropriate data structure (e.g. a heap).

If the same primitive transformations have multiple occurrences in the transformations, it is better to convert conflict to a memo function.

The main loop of the algorithms 1 and 2 can be terminated as soon as

\[
\max_{(i', j') \in \text{uncomputed}} \text{new}(\text{before}, i', j') = 0
\]

Performance of these algorithms

Algorithms 1 and 2 take in worst case \(O(n^2)\) global commutation computations (usually better). Algorithm 0 always has \(O(n^2)\) computations and \(O(n^3)\) operations to compute the transitive closure.

In order to compare the algorithms a small simulation study was performed. In these simulations a random conflict matrix was used in which two elements conflicted with a given probability \(p\) (i.e. a fraction of \(p\) entries of the conflict matrix was \(true\)).

The results are shown in the figures 7.1, 7.2, and 7.3.

When conflict can be computed cheaply, algorithm 0 is the obvious candidate. However, it can be expected that the cost of these computations is far from negligible. Algorithms
Figure 7.1: Transformation size versus number of commutation computations
The curves for Algorithms 1 and 2 overlap.

Figure 7.2: Transformation size versus number of accesses to before array
1 and 2 avoid many unnecessary comparisons. Algorithm 2 on the other hand is very expensive, while the complexity of algorithm 1 is similar to that of algorithm 0 \( O(n^3) \).

### 7.7 Step 2: Finding and solving conflicts

This step takes as input the raw blocks that were found in the first step, and locates the conflicts between the primitive transformations in each block. The first step has only used the global commutation behaviour of the primitive transformations. It is possible that two transformations that do not commute globally nevertheless commute locally for a large set of inputs.

The second step works on one raw block at a time, and determines the primitive transformations that are truly conflicting. This step also uses information about local commutation behaviour of the transformation.

In order to locate conflicts the merge tool examines all weaves of the primitive transformations that must be merged. A weave is a permutation of the union of the primitive transformations in both transformations in which the original order of the primitive transformations is respected. Thus in a weave \( T_{a,t} \) must occur before \( T_{a,t+1} \). Weaves are called interleavings in some other literature.

If we set out a grid with the primitive transformations for \( T_a \) on one axis and the primitive transformations for \( T_b \) on the other, every weave corresponds to a path in this grid.
A point in this grid represents a set of values, the origin represents the original value $X$ and the set of values at some other point are obtained by applying all transformations, that are represented by paths from the origin to this point, to the original value $X$. If the set of values at every point contains precisely one value, then the operations $T_a$ and $T_b$ commute. Conversely if $T_a$ and $T_b$ do not commute there is a point that contains more than one value. A point is called single-valued if its set of values contains one element, otherwise the point is multiple-valued.

The frontier set

Now we can take the set of multiple-valued points that have only single-valued points on all paths from the origin to this point, we call this set the frontier set. This frontier set gives useful information about which primitive transformations in $T_a$ and $T_b$ actually do conflict. The value set for a frontier point $z_{i,j}$ contains two different values: $T_a(i_{i-1,j})$ and $T_b(i_{j-1})$, thus $T_{A,i}$ and $T_{B,j}$ do not commute on $z_{i-1,j-1}$.

The information about the frontier set can help the user to decide how this conflict should be solved. A list of the decisions that a user can make was given in section 7.4. The first possible decision is to impose an ordering on the primitive transformations, i.e. to state that $T_{A,i}$ must always be performed before (or after) $T_{B,j}$. In this case the merge tool only considers weaves that satisfy this constraint. The other decisions change the original transformations. When a frontier point is “resolved”, a new frontier is defined. The new frontier is computed and presented to the user, until all conflicts have been resolved.

In many cases the primitive transformations in a compound transformation can be performed in a different order without affecting the outcome of the transformation, because the primitive transformations locally commute with each other. This can be used to move the frontier points away from the origin. When the frontier points are far removed from the origin, the user probably needs to make fewer decisions.

7.8 Computation of commutation behaviour

The algorithms in the previous section assume that there is some procedure to determine whether two transformations commute (locally or globally). This section describes several
approaches.

7.8.1 Computation of global commutation information
There is a trade-off here between speed and exactness of the comparison. The determination of global commuting properties must be conservative: the procedure must never tell that two operations commute globally when that is not true. However, it is not necessary to be very precise, when that is too expensive: the next step corrects any overly conservative estimates. When the procedure is too conservative the next step in the merging is more expensive of course.

Ways to determine global commutation information:

- Using user supplied hook functions. The user can define one or several pre-defined procedures that determine whether two transformations commute. These are then called by the merging process.
- Using information about read/write sets. A read set is the part of the OMS (i.e. a set of object-attributes and relation tuples) that is examined by the operation. Similarly, the write set is the part of the OMS that is modified by the operation. Two operation commute globally when the read set of one is disjoint with the write set of the other. One problem here is that it is usually hard to determine the read and write sets independently from the state of the OMS on which the operation is applied.
- An important special case is when an operation only reads or writes attributes in the receiver of the message. This information can be supplied by the user when declaring a new method. Thus the commutation information is made part of the type system.
- Using a special rule-base in which a user can declare which transformations commute, or a formal specification of the methods from which the system can deduce there commutation behaviour.

Of course several of these methods can be combined.

7.8.2 Computation of local commutation information
The same procedures as for global commutation can be used. There is one additional possibility:

- Execute the transformations and see whether the resulting OMS is the same.

7.9 Problems with operation-based merging
Replaying transformations can pose problems with respect to reproducibility of the results. Well-behaved primitive transformations are functional with respect to the OMS, i.e. the resulting OMS state is a (mathematical) function of the initial state and the arguments
of the transformation. Non well-behaved transformations are not replayable: on different occurrences they may result in different final states of the OMS when applied to the same initial OMS state. Not all operations on the OMS however are well-behaved. There are two main reasons why a primitive transformation is not well-behaved: user input and object creation. The result of a transformation that depends on user input, is not reproducible because the user input may vary from time to time. Object creation in CAMERA requires the generation of a unique object identifier. Thus when the same transformation is replayed multiple times, the objects may have a different object-identifier each time. The problems due to user interaction have been solved by making it illegal for methods to communicate with the user directly. All communication is performed by tools that are implemented outside the OMS. These tools then send messages to the OMS. The execution of these messages will not involve any user-interaction.

The problem with the creation of object-identifiers could potentially be solved in two different ways. The first method registers for a transformation which object-identifiers were created when this transformation was originally executed. When the transformation is replayed, objects with the same identifiers will be created. With this method complications arise when the replayed transformations create a different number of objects than they did originally. The second approach uses equivalence of the end-results instead of strict equality. In this case two OMSes are compared modulo the values of the object-identifiers. Two OMSes are considered equivalent iff there is a bijective mapping between their object-identifiers, that transforms them into each other. The CAMERA merge tool uses the latter approach.

Transformations can contain a number of primitive transformations that are redundant. If for two primitive transformations \( T_{a,i} \) and \( T_{a,j} \) it holds that \( T_{a,j}(T_{a,i}(x)) = T_{a,j}(x) \) the primitive transformation \( T_{a,i} \) can be omitted from the transformation. Examples of redundant transformations are operations on objects that are deleted later on, and update operations whose results are overwritten by later updates. Removing redundant transformations is attractive because it:

- speeds up conflict detection
- removes unnecessary conflicts

Operation-based merging depends on the re-usability of operations. The recorded operations must be replayable on other OMSes and should adequately represent the user’s intention. Different operations may have the same result in the original OMS, but behave differently under other circumstances. An edit operation on a text file can be modelled in different ways. The operation can be modelled as a value replacement of the entire file with its new value. This approach gives unsatisfactory results with respect to merging. Edit operations on the same object in different lines of development always lead to conflicts. A slightly better approach treats a file as a sequence of characters and uses operations that insert or delete bytes at certain positions in this file. Another approach uses operations that are based on modifications to lines in this file. This is the level at which most text merge tools operate. A more sophisticated approach uses a richer set of operations that also use
the structure of the text. A text may have an internal structure, e.g. a division in chapters and sections, or an even more complicated structure as a program source text. Operations that use this structure have a higher degree of re-usability, e.g. it is much more useful to record that a user has replaced the second paragraph of section 1.2 than to record that some bytes at a specific offset in the file have been modified.

7.10 Conclusions

Existing merge tools are either very crude, or very sophisticated but of limited applicability. Operation-based merging is an attractive alternative. It uses semantic knowledge about the objects and their operations. In addition, it is extensible. New object types and operations can be added and these can then be merged as well. Because operation-based merging uses the operations of a data-type it automatically maintains its data-type invariants.

A prototype of the merge tool has been implemented. It is currently used to evaluate the ideas that were presented in this chapter.
8 The CAMERA prototype

8.1 Overview

This chapter describes the architecture of the current Unix-based prototype of CAMERA. The prototype has been implemented using a mixture of C [KR88] and Elk Scheme [Lau90].

At the top level CAMERA consists of one or more Album OMSes and one or more environments. At each node in the loosely-coupled network there is at least one Album OMS, but if there are multiple completely unrelated activities at a node there can be several. Usually there is only one Album OMS per network node, that is shared among all CAMERA users at that node.

Every Album OMS has zero or more environments associated with it. Commonly each user has a private environment, but it is also possible for several users to share the same environment. The environment maintains the current state of the snapshot OMS. Complete states of the snapshot OMS are stored as snapshots, and represented as objects in the Album OMS.

The Album OMS is used to store information about the development process and is itself not subject to version control. The user can perform OMS operations on the Album OMS and on the snapshot OMS (via the corresponding environment). The Album OMS and the environment are able to communicate with each other, e.g. when new snapshots are created.

8.2 Process architecture

We now describe how the functionality of different parts of the CAMERA system is mapped onto different processes. A global overview of the processes involved is shown in figure 8.1.

A central design approach has been to create a separate process for each independent "resource". This has the advantage that all information about this resource is present in one process thus reducing interprocess communication. A further advantage is that this makes it possible to distribute the management of the different resources over different processors, which could give a better performance.
Each Album OMS is represented by one Album OMS process. The Album OMS process handles the execution of Album OMS messages, and the persistent storage of the contents of the OMS. Since the Album is global to several users, it is logical to create one process that maintains the state of the Album OMS and let other processes communicate with this process. Because there can be several different Album OMSes at one network node there can also be several Album OMS processes. Most operations only influence the state of the snapshot OMS, and not the Album OMS, and therefore it is a straightforward decision to split the Album OMS and snapshot OMSes into different processes.

Each environment is a resource that is largely independent from the Album OMS and the other environments. Therefore, every active environment is represented by an environment process. An environment process belongs to one Album process. When a user starts working with a specific environment, a request is sent to the Album OMS to get access to the environment process for this environment. If there is no active environment process for this environment, the Album process starts a new one. Likewise, when there are no longer any active users for an environment, the environment process is terminated, and its state is stored as a snapshot.

Every active user is represented by a user process that provides a command shell. This shell communicates with the environment process, and with the Album process. The command shell only communicates with the OMS by sending messages to objects in the OMS. The results of the execution of these messages is then returned to the command shell.

The command shell is also responsible for the interaction with the user. Methods defined in the OMS cannot perform I/O operations with the user. Interactive tools must be defined in the command shell, which can then access the OMS. This approach has important advantages for transformation logging. Methods that do I/O with entities outside the OMS, cannot be replayed as part of a transformation. If, however, user I/O is implemented by tools outside of the OMS, the corresponding transformations do not contain operations
that perform user I/O, and can therefore be replayed without problems. Successive snapshots tend to be very similar. This similarity is used to compress the size of snapshots, using the techniques described in chapter 4. These techniques store common parts of multiple snapshots only once. The maximum storage reduction is achieved if all snapshots on a network node are stored in one global snapshot storage. Managing this storage is the responsibility of the snapshot storage manager process.

The environment process interacts with both the Album process and the snapshot storage manager. The environment process caches (a part of) the contents of its current snapshot, and it is responsible for the execution of methods in the snapshot OMS. The environment process registers all modifications to its snapshot OMS as a transformation. When a new snapshot is created the environment process sends its current state to the snapshot manager, that stores it as a new snapshot. Furthermore, as part of this operation objects are created in the Album OMS to represent the new snapshot and transformation.

Interprocess communication is currently implemented by using TCP/IP [Com88]. Consequently, the prototype can be used both on both local area networks and wide area networks (e.g. the Internet).

The implementation also offers a Unix compatible interface (described in section 8.3) that is used to treat standard Unix programs as methods inside the OMSes. This makes it possible to re-use compilers etc. in CAMERA. Both OMSes can start up subprocesses for the execution of Unix tools. Special features are needed to make it possible to use Unix tools as part of the derivation management. The problem is that Unix tools normally have side-effects on the file system, and the methods that are used in a derivation must be side-effect free. Section 8.3.3 describes the solution that was chosen for CAMERA.

The OMSes must be able to handle large values, e.g. byte-strings that contain sources or binaries. Communicating these entire values among processes is inefficient. Furthermore, it is desirable to perform some form of delta-compression on these values. Therefore, the implementation contains a long-field manager that is described in section 8.4.

For the computation of derived values (see chapter 5) a cache of derived values is maintained. It would be possible to have a cache per environment (plus one in the Album OMS process), but then the contents of the cache cannot be shared among multiple environments. The caches of different environments usually show a large amount of overlap. This is certainly true when tools are handled as derived values from their sources. Many environments share the set of available tools, and by putting them in a shared cache they can be re-used. Therefore, CAMERA includes a cache manager process (see section 8.5) that is shared by all these environments.

8.3 Unix compatibility

This section discusses facilities that allow existing Unix software to run on top of the OMSes in CAMERA. Such facilities are a necessity since they allow the re-use of (a part of) the large existing body of Unix programs.
8.3.1 A file-system view of the OMS

Unix tools depend on the file-system and use operating system calls to access it. The OMS must thus be able to provide file-system-like behaviour for these tools. This is accomplished by adding special object types and relations to the OMS which have the semantics of traditional files and directories. Directory entries are mimicked by using the directory relation. A similar approach was taken in PCTE, [PCT]. It is not necessary that all objects in the OMS behave like files and directories. Objects that do not behave accordingly are invisible to the Unix tools.

8.3.2 Implementation of the file-system view

There are several possible approaches for interfacing standard programs that use file I/O to an OMS. One approach would be to replace the standard Unix system calls with calls to similar functions of the OMS, but this would require (source) modifications to the tools or to the operating system.

A different approach, that is implemented in the CAMERA prototype, uses a NFS file server [Now89]. A NFS file server is a process that provides file-system functionality. It was designed for sharing file-systems among multiple computers. The NFS protocol contains some 18 calls e.g. to read/write files, examine directories etc. File operations on files that reside on an NFS file system are translated by the client kernel into an NFS call that is then transmitted to the NFS server via a remote procedure call (RPC). This process is illustrated in figure 8.2. For the user this process is entirely transparent: a local file-system behaves the same as an NFS file-system. The file-system view on the CAMERA OMS is implemented by mapping NFS operations on OMS facilities. Thus the OMS acts as an NFS server. For all tools the OMS implements a standard Unix file system, without any modification whatsoever.

Normally the operations of these tools must be restricted to the OMS, and tools must be unable to access files outside of the OMS. Otherwise there are dependencies on objects that are outside the OMS arise, and this is not acceptable, especially for the snapshot OMS. This restriction is enforced by using the Unix chroot system call. The chroot system call changes the mapping of file-names to files for the executing process and its descendants. The call takes the name of a Unix directory as argument and maps the root of the file system onto this directory. Therefore, the process is only able to access the file tree that is rooted at this directory. Thus by using the chroot call a process is restricted to the file-system that is maintained by the CAMERA NFS server.

The Unix OS maintains for each process a certain amount of state that is not represented directly in the file-system, in particular the current working directory and the values of environment variables. This state information acts as an implicit input for the Unix programs and can therefore influence their behaviour. The current directory influences the mapping of file-names to the actual files. Environment variables can be read by the program and are frequently used for setting options. Because these pieces of state information act as inputs they must be controlled by CAMERA. Only in this way they can be subjected
to version control and used for dependency management. This is achieved by passing the working directory plus the environment variables as extra parameters to the methods that invoke these tools. **CAMERA** changes to the correct directory and sets the environment variables correctly for the process that actually executes the tool. A similar approach is used by Odin, that uses a special directory for tool execution, and that normally eliminates all environment variables from the environment of the process that executes the tool.

### 8.3.3 Side-effect-free behaviour

For the computation of derived values in **CAMERA** a strictly functional behaviour is essential, as was explained in chapter 5. These computations must not have any side-effect on the contents of the OMS. Many standard Unix tools can be described in a functional way. Currently, however, the implementations are not functional because they create and delete files in the file system.

For the purpose of using standard Unix tools in derivations **CAMERA** contains a *change layer*. The change layer provides a shadow image of the OMS for the tool. Changes that are made by the tool are not incorporated into the OMS but are only modified in the shadow OMS that is maintained by the change layer. The change layer is completely transparent for the tool, so the same tool can be used in both a functional and a procedural (with side-effects) way.
When the tool is finished, the change layer packages together all modifications that were made by the tool and returns these as the compound return-value of the tool invocation. For example, for the files that were created during a tool invocation, the change layer returns a set of 2-tuples (file-name, file-contents) that contain the names of the created files plus their contents. Temporary files that are created by the tools are not part of the result since they have been deleted when the tool is finished.

Since the change layer prevents all modifications to the OMS, the tool behaves as a side-effect-free method.

The change layer thus performs two different functions: it acts as a shadow OMS that prevents any changes to the real OMS, and it provides a return-value of the tool execution.

8.4 The long-field manager

Values of small data-types such as integers and short strings are easily communicated between different processes. For large data-types (e.g. the byte string that represents the contents of a file) this is too costly. These values are stored as long-fields. Instead of the entire long-field values processes exchange a long-field identifier that is used to access the corresponding value. Long-fields are immutable, thus a given long-field handle always refers to the same value. Operations on long-fields, that would ordinarily change the value of the long-field, return a new handle that refers to the new value of the long-field. The long-field manager offers an interface that performs standard file/string operations on long-fields. Similar to object-identifiers, long-field identifiers are globally unique. This property is used to optimise communication between different CAMERA systems, since it is easily determined whether a given long-field is present at a remote site by simply comparing long-field identifiers.

The long-field manager is also responsible for delta-compression of long-fields. When a new long-field is created it can be designated as a successor of another long-field. The long-field manager can then apply delta-compression techniques with respect to this predecessor.

8.5 The cache manager

The environment process maintains the current snapshot for an environment. It also caches derived values. But since multiple environments may contain the same derived values (e.g. when tools are modelled as derived values from their sources), there is also a global cache manager. If a derived value is requested from an environment, first the local cache is searched. When the value is not present in the local cache it is requested from the global cache. If this value is also not present in the global cache it is computed by the OMS process and then put in the local cache. When the user has indicated that this derived value must be cached globally it is added to the global cache as well. For some classes of derived values (e.g. easily computed methods, or methods with very complex inputs) the overhead
for communication with the global cache manager is too high. These are only cached by
the environment process. The user can indicate for each derived value (a combination of
an object plus a functional method on that object) how it is cached and re-computed.
The global cache manager identifies a derived value by a 4-tuple

\((\text{ObjectId}, \text{MethodName}, \text{Arguments}, \text{Inputs})\)

where the functional method \text{MethodName} is sent to \text{ObjectId} with \text{Arguments}. The
\text{Inputs} that are examined for the computation of this derived value are represented as a set
of 2-tuples \((\text{InputId}, \text{InputValue})\), where \text{InputId} identifies one input (that is an object
attribute or a derived value) and \text{InputValue} is the current value of this input.

8.6 Conclusions

The current prototype implements most of the features that have been described in this
thesis. Certain features are not fully operational at the moment of writing, e.g. merging
and the NFS server, but will be completed in the near future.
Current experience with the prototype is favourable. Even though the implementation is
not very optimised, the response times for normal operations are very acceptable.
9 Requirements revisited

The previous chapters have described the structure of the CAMERA system. This chapter compares the system with the requirements that were listed in the first chapter.

9.1 The repository

The repository in which data is stored forms a major aspect of any support system. The CAMERA repository is formed by a relatively simple, but powerful object-oriented database, that meets the requirement for being an expressive data storage. In this respect it is similar to most modern support environments that are frequently based on an object management system. Objects can have multiple attributes that describe their properties. The type mechanism is used to describe the internal structure of objects and their operations. Multiple inheritance helps re-using operations defined for one class on other classes. Relations are used to store references among objects.

Simpler object management systems, e.g. PCTE, only provide mechanisms to create and manipulate object types, but do not provide mechanisms to define operations on these objects. CAMERA does have this capability. New methods can easily be added to the system. This allows for interesting features, such as built-in derivation management, computed relations and triggers. The NFS server makes it possible to incorporate normal Unix programs, and treat them like methods in the OMS.

9.2 Version management

CAMERA has several unique features with respect to version management. Objects have world-wide global unique object-identifiers, which make it possible that objects maintain their identity during exchanges. Therefore, the identity is independent from naming structures that are defined on the objects. This is an advantage in loosely development environments because different participants are free to choose their own naming conventions.

The object-identifier can be used to trace the development of objects, by examining the value of the object in the snapshot history.

Each version of an object is uniquely identified by the snapshot in which it appears. Since snapshots are immutable, versions of the objects in snapshots are also immutable.
Integral version management records the state of an entire, self-contained workspace in the form of snapshots. This guarantees that history recording of the workspace is complete. It is impossible to accidentally omit important objects from a snapshot.

The CAMERA mechanisms can be used to undo the effects of recent operations. The operations of creating a new snapshot and of changing the current snapshot of an environment to an old snapshot are reasonably fast. Therefore, it is easy to roll-back the contents of an environment to a previous state. CAMERA also provides support for "redo"-ing selected operations. All user level operations are recorded as transformations. Transformations may be edited to remove harmful operations and subsequently they can be replayed on the original snapshot. Used in this way CAMERA provides a powerful undo/redo mechanism for the contents of the entire environment.

In CAMERA version management is completely transparent for tools in the workspace. Tools automatically use the versions of the objects in the current snapshot and need not be aware of the existence of other versions. In systems where version control is built on top of other facilities, this version transparency is much harder to achieve.

In traditional systems variant management is normally used for two different purposes: supporting parallel development and selecting sets of objects, that are functionally related. The first purpose is achieved more easily in CAMERA by creating an environment for each line of development. Currently CAMERA does not provide specific facilities for handling the second form of variant management. One of the reasons is that different types of objects normally require different forms of variant management. A programming language that has facilities for conditional compilation requires other variant management facilities than languages that do not have these facilities. Other types of objects such as pictures or documentation have entirely different requirements. Building variant management for specific object types is not a difficult task in CAMERA because it already provides most mechanisms. The main functionality of most variant management systems is that they provide ways to label variant objects and offer a query mechanism to select specific variants. CAMERA offers attributes and relations that can be used for labelling. Furthermore, it is easy to add methods that implement queries. The built-in derivation management and computed relations are also important building blocks for variant management.

### 9.3 Consistency management

Integral version management is an important aspect in helping the user select consistent sets of versions. All objects in a snapshot have actually been used together, and this gives us a reasonable degree of confidence that they form a consistent selection. When a snapshot contains inconsistencies, the developers normally attempt to remove these. In systems that force the user to explicitly select a set of versions, consistency is much harder to achieve.

New inconsistencies can arise when data is exchanged between environments. The CAMERA mechanisms for exchange, pieces and merging, help reducing such inconsistencies.
The objects in a piece all form part of the same snapshot, and are thus likely to be consistent with each other. Inconsistencies may arise during a transplant when needed objects have been omitted from the piece. The piece editor with a good rule-base prevents such omissions. The merge mechanism also contributes to consistency maintenance. Operation-based merging uses the operations that are defined on a data-type in the OMS. This form of merging avoids data-type inconsistencies because the operations automatically respect the invariants for this data-type. Additionally, operation-based merging provides important support in locating and solving conflicts.

Derivation management guarantees consistency between derived values and their inputs. The fact that derivation management facilities are built-in has several advantages with respect to most previous systems. Unlike systems such as Make the CAMERA derivation manager need not be invoked explicitly to re-compute derived values. Furthermore, the system automatically detects all input dependencies for derived values, and there is no need for a separate description of a derivation such as a Makefile. Therefore, the system automatically adapts itself as the structure of the derivations is changed, e.g. because new inputs have been added. Derived values are well integrated with other operations in the system. When a method calls another function there is no distinction whether that function is a derived value or not. Derivation management makes the input dependencies of a derived value available in the form of the depends-on relation, that amongst others provides important input for the formation of pieces.

9.4 Cooperation

One of the requirements for parallel development is that different lines of development should normally be isolated from each other to avoid interference. In CAMERA different lines of development are normally mapped onto different environments, that are automatically isolated. Each environment provides a complete, private workspace.

Several mechanisms are provided by CAMERA for the exchange of data among different lines of development. A first mechanism is the exchange of entire snapshots. This is useful when the entire state of a workspace must be exchanged. It can be used when a new line of development starts from a base-line snapshot or when further development continues on a different computer.

Pieces are used to exchange parts of snapshots. Pieces are very flexible: they can be used to communicate arbitrary objects/relationships. Because pieces are stored persistently, they can be re-used to transmit new versions of the objects in the piece. The piece editor aids in constructing/maintaining pieces. It has its own rule-base that can be extended for new object types.

The CAMERA approach to merging is operation-based. It attempts to merge the operations that were used in the different lines of development. Operation-based merging is made possible because CAMERA logs the operations that are being performed. One advantage of operation-based merging is that it can merge arbitrary objects, even for
user-defined classes. Thus operation-based merging is able to merge entire snapshots. A second important advantage of operation-based merging is that it re-uses the operations of an abstract data type, and therefore maintains the data-type invariants.

9.5 Distribution

CAMERA is different from most other distributed systems in that it was designed for use in a loosely-coupled network. In such a network it is physically impossible to hide distribution. Therefore, distribution is visible in CAMERA. For normal operations, developers only work in their own environment. Environments themselves are not internally distributed. Therefore, work inside the workspace can continue even when the network is disconnected. This also simplifies the design of tools that run inside the workspaces, because they don’t have to handle exceptions that arise when certain objects are not locally available.

When data is communicated among environments the current status of the network becomes visible, because the receiving environments may not currently be reachable. Such communication requests are queued if they cannot be executed immediately. CAMERA executes these communication requests when the receiving environment becomes available. The communication is optimised so that it can also be used in slow networks.

9.6 Miscellaneous

The sparse index techniques from chapter 4 are used to store snapshots very efficiently. Therefore, CAMERA is more efficient in this respect than other systems where version management facilities are built on top of the OMS. Furthermore, these techniques make it possible to create new versions (new snapshots) very frequently.

CAMERA is a very general purpose system that is not tailored towards specific applications. The system can easily be extended to handle new data-types and applications.

CAMERA does not contain hard-wired policies, but supplies mechanisms to implement such policies. The system can be used to automate recurring tasks, by adding new methods and triggers.

9.7 Conclusions

Summarising, CAMERA meets most of the requirements that were outlined in the first chapter. Its most interesting features are its approach to integral version management, derivation management, loosely-coupled distribution and the mechanisms for exchange: pieces and merging.
Summary

Introduction

The use of computers for many kinds of activities is still rapidly increasing. Frequently these activities concern a form of cooperative work where users exchange their results. Examples are co-authoring documents and software development. In the past such activities were ordinarily performed on one central computer, but at present it is common that participants work on different computers and exchange results between these computers. Two different situations can be distinguished with respect to the physical distribution of these computers. In a tightly-coupled network all computers are connected via high-speed links with a high availability (e.g. a LAN). Working in a tightly-coupled network is very similar to working with one central mainframe. The opposite is a loosely coupled network where links between computers can be slow or unavailable. Loosely-coupled networks can be found in situations where the participants work on different locations, or when participants want to continue their work at places where permanent network connections are not available, e.g. at home or while travelling.

Hardly any system for computer supported cooperative work has been designed for use in a loosely-coupled distributed environment. This observation was the motivation for the CAMERA project: the design of a system to support cooperative work in a loosely-coupled distributed environment. This thesis describes the system that is the result of this project, and of which a prototype has been implemented.

The architecture of CAMERA

One of the main functions of CAMERA is to act as a repository for the product (e.g. a document or a program) that is being developed. Users have workspaces that contain the versions of the product on which they are currently working. Such a workspace is called an environment in CAMERA. Environments have the following characteristics:

First of all environments are not internally distributed. This may sound a little strange for a distributed system, but CAMERA was designed for an situation where connections can be frequently unavailable. When the data in the workspace itself would be distributed, certain parts of the workspace could become unavailable when the network is partitioned. All applications that run inside the workspace would have to handle the problems that occur
when certain parts of the workspace are not available. These problems can be prevented by copying the relevant parts of the workspace to a local computer. In larger projects the workspace will contain many components. Selecting and copying these components manually is cumbersome and error-prone. In CAMERA the system takes care of this process.

A second property of environments, that is closely related to the previous one, is that environments are self-contained, they contain all data and programs that are needed for development activities. The contents of an environment does not contain references to components outside this environment. This implies that transferring the contents of an environment to a different computer becomes much easier. It is not possible to accidentally forget components or tools.

A third property of environments that different environments are completely disjoint. Modifications of the contents of one environment do not affect the others. There are two reasons why this is desirable. The first reason is technical, when two environments are located on different sites in a loosely-coupled, it cannot be guaranteed that they can always communicate immediately. The second reason is that very frequently participants want to be isolated from the activities of others. Problems arise when multiple participants attempt to edit the same document or when a programmer modifies a program while another attempts to debug it.

**History recording**

Recording the development history is important for determining the nature of the changes in a certain line of development. Most older version control systems only record versions of individual components. In CAMERA versions are simultaneously recorded for all entities in the workspace. This approach is called integral version management. Versions of the contents of an environment are stored as snapshots.

Snapshots are in some respects similar to the backup copies (or dumps) that are found in other systems. However, they have several advantages. In traditional systems, access to older backups is frequently cumbersome. Backup’s are usually stored on tape, so fast random access is not possible. Another complication arises with incremental backups, that only contain data that was modified since the previous backup. When a user wants to retrieve a set of old versions of components it may be necessary to examine several incremental backups. In CAMERA transparent access to old snapshots is possible due to the data-structures that are described in chapter 4. Furthermore, snapshots are self-contained thus a complete set of old versions that have been used together are always contained in the same snapshot. A further advantage of snapshots is that they are explicitly represented themselves as objects (in the Album OMS). Therefore, it is easy to store information about the snapshots and their relations by using the OMS mechanisms.

Apart from the snapshots, there is another mechanism in CAMERA to record the development history called transformation. Basically, a transformation is a log of the operations
that were performed by the user between one snapshot and its successor. Because all operations in CAMERA are modelled as message sends in the OMS, transformations are a list of the top-level message sends that were executed by the user. Transformations can be replayed on a different snapshot. The message sends are then executed. Transformations can also be edited before replaying, e.g. to remove undesirable operations. The combination of snapshots and transformations can therefore be used as a powerful undo/redo mechanism.

The OMS

Similar to most of the recent systems for cooperative work, CAMERA uses an Object Management System as basic repository instead of the more traditional file-system. OMSes offer several advantages such as typing, the possibility to record information about the relations among components and the ability to define new operations.

CAMERA contains two different types of Object Management Systems: the Snapshot OMS and the Album OMS. The Snapshot OMS is used to store the data in the user's workspace. The contents of the Snapshot OMS are version-controlled, and historical versions of each workspace are stored as snapshots. The Album OMS is used to record data about the development process. Environments and snapshots are represented as objects in the Album OMS. The Album OMS is not subject to version control.

Both the Album OMS and the Snapshot OMS have a very similar data model, they both belong to the group of object-oriented data-bases. The main difference from the user's point of view lies in the pre-defined set of types and operations. The term OMS will be used when describing common features.

The CAMERA OMS belongs to the class of object-oriented data-bases. Entities are modelled as objects. An object has an object-identifier that provides a unique identification of this object. Each object belongs to a class, that describes the attributes of its instances.

A class can have multiple super-classes from which they inherit attributes and operations. Operations are modelled as methods of which there are two different kinds: functions that have no side-effects on the contents of the OMS and procedures that can have side-effects. Methods are associated with a class. When an operation is invoked on an object (this is called a message send) the system locates the corresponding method in the class of this object or in one of its super-classes.

The OMS also provides relations to describe the interrelations among objects. Essentially a relation is a named set of relationships (n-tuples). Relations are represented as objects, and relations can be queried by sending a message to the corresponding objects. Note that individual relationships are not modelled as objects. In several object-oriented data-bases, the attributes of an object can contain either a primitive value such as an integer or string, or a reference to another object. In CAMERA the attributes can only contain primitive values, references between objects must be modelled as relationships in an appropriate relation. In systems that use instance variable to record references among objects it is
usually difficult to discover which objects refer to a given object. Because relations are bi-directional such queries can be answered easily in the CAMERA OMS. The OMS is extensible, new classes, relations and methods can be added by the user.

It is customary in data-bases to make a distinction between data and schema information (also called meta-data). The data-base schema describes normally the available types and relations. In most object-oriented data-bases schema information (such as the available classes and methods) is represented in the form of objects. For example, each class is represented by a class object. This approach was also used in the CAMERA OMS. Therefore, there is no need to have a separate data definition language, the schema can be changed by sending message to the appropriate (meta-) objects. Furthermore, the same mechanisms that are used to exchange ordinary objects can also be used to exchange schema information.

**The Album OMS**

As we have seen the Album OMS contains data about the development process. Individual workspaces are modelled as objects that are called *environments*. When a user want to start working with an environment an activation message is sent to the environment object. Snapshots are also represented by objects, and there is a relationship between an environment and its most recent snapshot. Old snapshots can be loaded into an environment and are then used as the contents of the workspace. Modifications that are made in an environment are recorded as transformations, and the modified state of an environment can be recorded as a new snapshot. New snapshots are recorded automatically when a user stops working with an environment but it is also easy to make intermediate snapshots, by sending an appropriate message to the environment object. This process can also be automated.

When a new snapshot is created its predecessor snapshots are recorded in a special OMS-relation. In a similar way the corresponding transformations are recorded.

**Implementation of snapshots**

Each snapshot conceptually contains a version of all objects that were present in a workspace at a certain point in time. It is obvious that an implementation of snapshots that contains a copy of all objects that are part of this snapshot is extremely wasteful. Chapter 4 describes data-structures that can be used to implement snapshots much more efficiently. These data-structures are based on the observation that successive snapshot are normally very similar; only a few objects have been changed between these snapshots. Therefore, it is advantageous to store the contents of objects only once and let the snapshots contain references to these values. This is done by maintaining a special index. To find a version of an object in a given snapshot the object identifier of this object plus an identification of the snapshot must be given. The size of the index is further reduced by omitting repeated
entries. When an entry in the index refers to the same value as the entry that immediately precedes it in the index it is omitted.

Chapter 4 also describes data-structures that can be used to implement the storage of multiple versions of a set, that can also be used to implement versioning of relations.

Together, these data-structures provide efficient use of storage space, and give access performance that is similar that in the non-version-controlled case.

Derivation management

Derived values are values that can be computed automatically from a set of inputs in the OMS. An example of a derived value is a compiled program that can be produced by calling the compiler on its sources. Frequently, it is desirable that the derived value corresponds with its inputs. Thus, when one of the inputs is changed the derived value will have to be recomputed. In large systems with a complex structure of the derivations, it is cumbersome to handle this manually.

The derivation management facilities in CAMERA help automating this task. Derived values are modelled as the results of functional (i.e. side-effect-free) methods on specific objects. In traditional derivation management one of the big problems is in the determination of the total set of inputs for a given derived value. This information is absolutely necessary for determining whether a derived value is still up-to-date. The traditional solution is to either force the user to list all inputs, which is cumbersome and error-prone, or to develop specialised tools that determine the inputs, which is not easily extensible for new types of derivations. Because derivation management is tightly integrated with the OMS, the inputs for each derived value can be determined automatically. Basically, the derivation manager monitors all accesses to the OMS during the computation of a derived value. Every part of the OMS that is accessed is an input for this derived value. Because the Snapshot OMS is self-contained, derived values cannot use any inputs that are outside the Snapshot OMS. Therefore, derivation management can guarantee that all inputs for a specific derived value are known. Derived values are stored in a cache. When the user requests a specific derived value, the cache is examined for an up-to-date value. If none can be found the value is recomputed.

From the point of view of the user there is no real difference between a specific message send that represents a derived value and a normal message send. The main difference is in execution speed, using a derived value will be faster when the result can be found in the cache.

The information about the inputs for a derived value can be valuable to the user. For example, it could be used to determine the total set of source files for a given (compiled) program. This information is available to the user by a special depends-on relation.
Pieces

Normally when users exchange data, they exchange multiple objects (and relationships). **CAMERA** contains a mechanism called *piece* to describe such a group of objects and relationships. Pieces can be used to combine arbitrary collections of objects and relationships. Pieces can overlap: the same object (relationship) can be part of many pieces. In some aspects pieces are a generalisation of complex (or composite) objects that are found in other systems. In these systems there is normally only one fixed composition hierarchy. In **CAMERA** pieces can be used to define multiple composition hierarchies, that can be independent from the relations of its constituent objects.

Pieces are exchanged between snapshots by the transplant operation. During a transplant the old contents of the piece is overwritten by the new contents.

Pieces are constructed by using the *piece editor*. The piece editor contains a rule-base with rules that describe what objects/relationships are normally included in the piece. For example, when a program module is added to a piece, the rules can be used to describe that the modules on which it depends should by default be added, too. It is also possible to specify exceptions to a rule, for example that modules from a standard library should not be included by default. Although the piece editor makes suggestions about the composition of the piece, it is up to the user to make the final decision. The *depends-on* relation that is produced by derivation management gives important information that can be used by the piece editor.

Merging

Another type of support for the integration of the results of different lines of development, is the merging facility. The type of merging in **CAMERA** is called operation-based merging. Traditional merging can be characterised as state-based, only the initial and final states of development are used in the merging process. Operation-based merging uses the operations that were used in each development line (and that have been recorded as transformations) to determine the merged result. A basic notion here is that of commutation of operations that have been performed. Two operations are said to commute, when the end-result of first executing the first operation and then the other one gives the same end-result as when the order of execution is reversed. If all operations from the two different lines of development commute, the end-result of applying the transformations is independent from the order in which these transformations are applied. In this case the end-result is the result of the merging operation. However, conflicts may arise when the operations do not commute. In this case frequently there is only a small subset of both transformations that is causing problems. In operation-based merging these operations are detected and presented to user who must then solve the conflict.
Samenvatting

Inleiding
Het gebruik van computers neemt nog steeds toe voor een breed spectrum aan activiteiten. Vaak worden computers gebruikt voor activiteiten waarbij samenwerking centraal staat. Voorbeelden zijn het gezamenlijk schrijven van een boek of rapport of het ontwikkelen van programma's. In het verleden werden dit soort activiteiten vaak uitgevoerd op één centrale computer, maar het komt steeds vaker voor dat deelnemers werken op verschillende computers en resultaten moeten uitwisselen tussen die computers. Met betrekking tot de distributie infrastructuur kunnen grofweg twee soorten situaties onderscheiden worden. In een "closely-coupled" netwerk zijn alle computers permanent verbonden via een snelle verbinding. In feite is de situatie in zo'n netwerk vergelijkbaar met het werken op één centrale computer. In een "loosely-coupled" netwerk zijn de verbindingen tussen computers langzaam of vaak niet beschikbaar. Dit soort netwerken ontstaan vaak wanneer de deelnemers zich bevinden op geografisch verschillende locaties, of wanneer deelnemers willen werken op plaatsen waar geen permanente verbindingen zijn, zoals thuis of tijdens reizen.

Er zijn maar weinig computer systemen die het werken in een "loosely-coupled" netwerk ondersteunen. Deze observatie vormde de belangrijkste motivatie voor het doel van het CAMERA project: de ontwikkeling van een systeem voor computer-ontwerde samenwerking in een "loosely-coupled" netwerkomgeving. Dit proefschrift beschrijft het systeem dat het resultaat is van dit project, en waarvan een prototype gebouwd is.

De architectuur van CAMERA
Eén van de belangrijkste functies van CAMERA is om te dienen als een opslag voor de produkten (bijv. documenten of programma's) die ontwikkeld worden. Gebruikers hebben een eigen werkruimte die ze kunnen gebruiken om hun eigen versies van produkten op te slaan. Zo'n werkruimte wordt environment (omgeving) genoemd. Environments hebben de volgende eigenschappen.

Ten eerste, environments zijn zelf niet gedistribueerd over meerdere computers. Op het eerste gezicht mag dit misschien wat merkwaardig lijken voor een gedistribueerd systeem, maar CAMERA is speciaal ontworpen voor het gebruik in een situatie waar netwerkverbindingen niet betrouwbaar zijn. Als de inhoud van een environment verdeeld zou zijn
over meerdere computers, dan zouden bepaalde delen van de environment onbereikbaar kunnen zijn als de verbindingen wegvallen. De gebruiker en alle programma's die met de inhoud van de environment werken zouden er voortdurend rekening mee moeten houden dat bepaalde onderdelen onbereikbaar kunnen zijn. Dit soort problemen kan voorkomen worden door er voor te zorgen dat de volledige inhoud van een environment beschikbaar is op een lokale computer. In grotere projecten zullen produkten uit een groot aantal componenten bestaan. Het is lastig om voortdurend alle componenten die een gebruiker nodig heeft met de hand te moeten selecteren en vervolgens te copiëren naar de lokale computer. **CAMERA** ondersteunt dit proces.

Een tweede eigenschap van environments, die nauw samenhangt met de vorige is dat environments “self-contained” zijn. Zij bevatten alle gegevens en programma's die nodig zijn voor ontwikkelactiviteiten. De inhoud van een environment bevat geen verwijzingen naar objecten buiten deze omgeving. Dit betekent dat het transporteren van de inhoud van een environment van de ene computer naar de andere eenvoudiger wordt, omdat het niet mogelijk is om componenten of programma's te vergeten.

Een derde eigenschap van environments is dat verschillende environments niet overlappen. Wijzigingen in één omgeving zijn niet zichtbaar in andere omgevingen. Er zijn twee redenen voor deze ontwerpbeslissing. De eerste reden is technisch, als twee omgevingen op verschillende computers draaien in een loosely-coupled netwerk dan is het niet altijd mogelijk om de wijzigingen in de ene environment direct over te sturen naar de andere. Een tweede reden is dat het vaak wenselijk is dat gebruikers geïsoleerd worden van de storende effecten van de activiteiten van anderen. Problemen kunnen ontstaan als meerdere gebruikers proberen om hetzelfde document te wijzigen, of als een programmeur probeert de fouten in een programma te vinden terwijl anderen tegelijkertijd bezig zijn om dit programma te wijzigen.

**Registratie van ontwikkelingen**

Het is belangrijk dat de ontwikkelingsgeschiedenis van een produkt goed wordt vastgelegd. Oudere versiebeheersystemen zijn vaak gericht op het bijhouden van de geschiedenis van individuele componenten (files). In **CAMERA** wordt de geschiedenis van de volledige inhoud van een environment bewaard. Versies van de volledige inhoud van een environment worden bewaard als snapshots.

Snapshots lijken in een aantal opzichten op dumps of backup copieën zoals die in andere systemen worden aangetroffen. Er zijn echter een aantal belangrijke verschillen. In traditionele systemen zijn oude backups vaak slecht toegankelijk. Backups zijn meestal opgeslagen op een extern medium zoals tape waarbij snel opzoeken lastig is. Extra problemen ontstaan bij incrementele backups, waarbij alleen de files die gewijzigd zijn sinds de vorige backup worden opgeslagen. Als een gebruiker een bepaalde historische versie wil terugvinden dan is het vaak nodig om een aantal backups te doorzoeken. In **CAMERA** zijn oude snapshots ook snel toegankelijk door speciale datastructuren die beschreven zijn in hoofdstuk 4. Omdat snapshots self-contained zijn, bevatten zij dus een versie voor alle
objecten die tezamen gebruikt zijn. Een andere voordeel van snapshots boven de traditionele backups is dat het eenvoudig is om extra informatie over de inhoud van snapshots en hun onderlinge relaties te registreren door gebruik te maken van de faciliteiten van het object management systeem.

Behalve snapshots is er nog een tweede mechanisme, de transformaties, om de ontwikkelingsgeschiedenis te registreren. Grofweg gezegd is een transformatie een lijst van de operaties die zijn uitgevoerd tussen een snapshot en zijn opvolger. Omdat alle operaties in CAMERA gemodelleerd zijn als de executie van methoden in het OMS, is een transformatie een lijst van alle methode-aanroepen die uitgevoerd zijn door de gebruiker. Transformaties kunnen opnieuw worden afgespeeld op een ander snapshot en alle methode-aanroepen worden dan uitgevoerd op dit snapshot. Het is ook mogelijk om voor het afspelen de transformaties te wijzigen, bijvoorbeeld om ongewenste operaties te verwijderen. De combinatie van snapshots en transformaties kan daardoor gebruikt worden als een krachtig undo/redo mechanisme.

**Het Object Management Systeem**

Net als de meeste andere recente systemen voor coöperatief werk, gebruikt CAMERA een Object Management Systeem (OMS) voor de opslag van produkten i.p.v. een gewoon file systeem. OMSen bieden een aantal voordelen, zoals typering, de mogelijkheid om relaties tussen objecten te leggen en om nieuwe operaties op die objecten te definiëren.


Het Album OMS en het Snapshot OMS hebben een vergelijkbaar data model, ze behoren beide tot de klasse van object-georiënteerde databases. Vanuit de optiek van de gebruiker ligt het grootste verschil tussen beide in de verzameling voor-gedefinieerde typen en operaties. De term OMS wordt gebruikt om de gemeenschappelijke eigenschappen te beschrijven.

Analoog aan andere object-georiënteerde databases worden entiteiten in het OMS worden gemodelleerd als objecten. Elk object heeft een object-identifier die dit object uniek identificeert. Elk object behoort tot een class, die de attributen van zijn instanties beschrijft. Een class kan een of meer super-classes hebben, waarvan het de eigenschappen erf. Operaties worden beschreven door methoden waarvan twee verschillende soorten bestaan: functionele methoden hebben geen zij-effecten op het OMS, en procedures die wel zij-effecten kunnen hebben. Methoden zijn geassocieerd met een bepaalde class. Als een operatie wordt aangeroepen op een bepaald object dan lokaliseert het systeem de overeenkomstige operaties in de class van dit object of een van zijn super-classes.
Het OMS bevat ook *relaties* om de verbanden tussen objecten te beschrijven. In essentie is een relatie een verzameling relationships (n-tuples). Relaties worden geregpresenteerd als objecten en queries op de relaties kunnen worden uitgevoerd door het aanroepen van methoden op deze objecten. Merk op dat individuele relationships geen objecten zijn maar gewone waarden. In sommige object-georiënteerde databases is het mogelijk dat de attributen van een object kunnen verwijzen naar andere objecten. In **CAMERA** kunnen attributen alleen maar primitieve waarden, zoals getallen of strings, bevatten. Referenties tussen objecten moeten gemodelleerd worden als relaties. In systemen die hiervoor attributen gebruiken is het vaak ingewikkeld om uit te vinden welke objecten refereren naar een bepaald object. In **CAMERA** kunnen dit soort vragen makkelijk beantwoord worden, omdat relaties bi-directioneel zijn.

Het OMS is uitbreidbaar, nieuwe classes, relaties en methoden kunnen door de gebruiker worden toegevoegd.

Het is gebruikelijk in data-bases om een onderscheid te maken tussen de gewone data en schema informatie (ook wel meta-data genoemd). Het data-base schema beschrijft normaal gesproken de beschikbare typen en relaties. In de meeste object-georiënteerde data-bases wordt schema informatie (zoals de beschikbare classes en methoden) geregpresenteerd als objecten. Deze aanpak is ook gekozen in **CAMERA**. Er is geen aparte schema taal, schema informatie kan aangepast worden door het aanroepen van methoden op passende (meta-)objecten. Verder kunnen de mechanismen die gebruikt worden voor het uitwisselen van normale objecten ook gebruikt worden voor het uitwisselen van schema informatie.

**Het Album OMS**

Het Album OMS bevat een aantal specifieke classes en relatie die niet aanwezig zijn in het Snapshot OMS. Individuele werkruiinten zijn geregvolumeerd als environment-objecten. Een gebruiker kan een environment activeren voor gebruik door het aanroepen van een methode op de corresponderende object in het Album OMS. Snapshots worden ook geregpresenteerd als objecten, en er is een speciale relatie die voor iedere environment het meest recente snapshot bijhoudt. Oude snapshots kunnen in een environment geladen worden, en worden dan gebruikt als inhoud van de werkruiint. Wijzigingen die gemaakt worden binnen een environment worden geregistreerd in de vorm van transformaties, en de gewijzigde inhoud van een environment kan opgeslagen worden als een nieuw snapshot. Nieuwe snapshots worden automatisch gemaakt als een gebruiker stopt met het werken in een bepaalde environment, maar het is ook mogelijk om tussendoor snapshots te registreren door een methode aan te roepen van het environment object. Dit proces kan ook geautomatiseerd worden.

Als een nieuw snapshot wordt aangemaakt, dan worden zijn voorgangers automatisch geregistreerd in een speciale Album OMS-relatie. Op een soortgelijke manier worden ook de bijbehorende transformaties geregistreerd.
De implementatie van snapshots

Conceptueel gezien bevat een snapshot versies van alle objecten die aanwezig waren in de werkruimte op het moment dat dit snapshot werd aangemaakt. Het is duidelijk dat een implementatie die afzonderlijke copieën opslaat van al deze objecten erg verspillend is. In hoofdstuk 4 worden een aantal data-structuren beschreven die gebruikt kunnen worden om snapshots efficiënt te implementeren.

Deze data-structuren zijn gebaseerd op de observatie dat opeenvolgende snapshots veel op elkaar lijken; slechts een relatief klein aantal objecten zal gewijzigd zijn ten opzichte van het vorige snapshot. Daarom is het voordelig om de waarden van (versies van) objecten slechts één maal op te slaan en om dan de snapshots te laten verwijzen naar deze waarden. Dit wordt gedaan door gebruik te maken van een speciale index. Om een versie van een object te vinden moet de object-identifier plus een identificatie van het snapshot gegeven worden. Deze index kan nog verder verkleind worden door repeterende items in de index weg te laten. Als een item verwijst naar dezelfde waarde als de voorafgaande kan hij weggelaten worden.

In hoofdstuk 4 worden ook een aantal data-structuren beschreven die gebruikt kunnen worden om versies van verzamelingen efficiënt op te slaan. Deze data-structuren kunnen ook gebruikt worden voor het implementeren van versies van relaties.

Tezamen leiden deze data-structuren tot een efficiënt gebruik van opslagruimte met toegangssnelheden die vergelijkbaar zijn met die van soortgelijke systemen zonder versiebeheer.

Derivation management

Afgeleide waarden (derived values) zijn waarden die automatisch berekend kunnen worden aan de hand van bepaalde inputs uit het OMS. Een voorbeeld van een afgeleide waarde is een vertaald programma dat automatisch geproduceerd kan worden door de vertaler los te laten op de tekst van dit programma. Het is vaak wenselijk dat een afgeleide waarde correspondeert met zijn inputs. Dus als een van de inputs wordt gewijzigd dan is het nodig om de afgeleide waarde opnieuw te berekenen. In grote systemen met een complexe structuur is het lastig om met de hand te moeten bijhouden welke afgeleide waarden opnieuw moeten worden berekend.

Het derivation management in CAMERA helpt bij het automatiseren van dit proces. Afgeleide waarden worden gemodelleerd als het resultaat van functionele (zonder zij-effect) methoden op bepaalde objecten. In traditionele systemen voor derivation management is het vaak een lastig probleem om de totale verzameling van alle inputs voor een afgeleide waarde te bepalen. Deze informatie is absoluut noodzakelijk om te kunnen bepalen of een afgeleide waarde nog wel up-to-date is. De traditionele oplossing is of om de gebruiker deze informatie te laten geven, wat ingewikkeld en foutgevoelig is, of om een speciaal programma de inputs te laten berekenen, wat slecht uitbreidbaar is voor nieuwe soorten afgeleide waarden. Omdat in CAMERA het derivation management
geïntegreerd is met het OMS, is het mogelijk om alle input afhankelijkheden automatisch te bepalen. Ruwweg bewaakt de derivation manager alle lees-operaties op het OMS tijdens de berekening van een afgeleide waarde. Alle onderdelen van het OMS die gelezen worden vormen een input voor deze afgeleide waarde. Omdat het Snapshot OMS self-contained is, kan de berekening van deze afgeleide waarde niet afhangen van objecten buiten dit OMS. Daardoor kan het derivation management garanderen dat alle inputs bekend zijn.

Afgeleide waarden worden opgeslagen in een cache. Als de gebruiker een afgeleide waarde opvraagt, wordt eerst gekeken of er nog een passende waarde in de cache zit. Zo niet dan wordt de afgeleide waarde opnieuw berekend.

Vanuit de optiek van de gebruiker is er geen wezenlijk verschil tussen de aanroep van een functionele methode die een afgeleide waarde representeert en de aanroep van een "gewone" methode. Het enig merkbare verschil zit in de executiesnelheid: het gebruik van een afgeleide waarde is sneller als het resultaat in de cache gevonden kan worden.

De informatie over de inputs van een afgeleide waarde kan waardevol zijn voor een gebruiker. Bijvoorbeeld kan deze informatie gebruikt worden om de volledige verzameling van bron-teksten voor een vertaald programma te bepalen. Deze informatie wordt geregistreerd in een speciale OMS relatie, de depends-on relatie.

Pieces

In een systeem voor computer-ondersteunde samenwerking zijn mechanismen voor de uitwisseling van gegevens tussen gebruikers belangrijk. Een primitief mechanisme voor deze uitwisseling in CAMERA is de uitwisseling van snapshots. Echter, in de meeste situaties levert dit niet het gewenste resultaat, omdat in een environment slechts één snapshot tegelijkertijd gebruikt kan worden. Het is dus niet mogelijk om op deze manier de gegevens uit meerdere snapshots te integreren.

Normaliter, zullen gebruikers bij het uitwisselen van gegevens een aantal objecten (en relationships) uitwisselen. CAMERA biedt een mechanisme, de pieces, om zo'n verzameling van objecten en relationships te beschrijven. Pieces kunnen gebruikt worden om arbitraire verzamelingen objecten en relationships te combineren. Verschillende pieces kunnen overlappen, een object (relationship) kan deel uitmaken van verschillende pieces. In bepaalde opzichten kunnen pieces gezien worden als een generalisatie van de samengestelde objecten die in andere systemen gevonden worden. In deze systemen is er meestal één onveranderlijke samenstellingshierarchie. In CAMERA kunnen pieces gebruikt worden om meerdere samenstellingshierarchiën te definiëren, die onafhankelijk kunnen zijn van de relaties tussen de objecten.

Pieces worden uitgewisseld tussen snapshot door de transplantatie-operatie. Als resultaat van deze operatie wordt de oude waarde van een piece vervangen door de nieuwe waarde. Pieces worden geconstrueerd met behulp van de piece-editor. De piece-editor gebruikt een verzameling regels die beschrijven welke objecten/relationships toegevoegd moeten worden aan het piece. Bijvoorbeeld als een module van een programma in het piece zit
dan is het waarschijnlijk dat andere modules die door de module gebruikt worden, ook toegevoegd moeten worden. Het is ook mogelijk om uitzonderingen op een regel te kunnen maken. Bijvoorbeeld om te beschrijven dat standaard modules uit een bibliotheek niet in het piece gestopt hoeven te worden. De piece-editor suggereert een bepaalde samenstelling van het piece, de gebruiker heeft het laatste woord. De depends-on relatie die door derivation management wordt bijgehouden levert belangrijke informatie voor de piece-editor.

**Merging**

Een andere vorm van ondersteuning van de integratie van resultaten in CAMERA is *merging*. De gebruikte techniek voor merging is operation-based merging genoemd. Traditionele systemen voor merging kunnen worden gekarakteriseerd als toestands-gebaseerd: alleen de initiële toestand en het eindresultaat van de ontwikkelingslijnen worden gebruikt. Operation-based merging is gebaseerd op de operaties die zijn uitgevoerd in een bepaalde ontwikkelingslijn (en die zijn geregistreerd in de vorm van transformaties). Een centrale notie bij operation-based merging is die van het commuteren van twee operaties. We definieeren dat twee operaties commuteren als het eindresultaat van het uitvoeren van beide operaties onafhankelijk is van de volgorde waarin de operaties zijn uitgevoerd. Als alle operaties uit de transformatie met elkaar commuteren dan is het resultaat van het uitvoeren van de transformaties dus onafhankelijk van de volgorde waarin de transformaties worden uitgevoerd. In dit geval kan dit eindresultaat gebruikt worden als eindresultaat van het merge-proces. Conflict en ontstaan als niet alle operaties commuteren. In dit geval is er echter meestal maar een kleine deelverzameling van operaties die echt problemen oplevert. Deze conflicterende operaties worden opgespoord en aan de gebruiker getoond die dan de conflicten moet oplossen.
Curriculum Vitae

25 April 1959 Geboren in Den Haag.


References


References


[NSE88] Nse 1.0 announcement., May 88.


References


Index

Adele 20, 29, 31, 43
Album OMS 35, 44, 66
process, 146
arc 111
attribute 52, 54
name, 52
template, 55
type, 52, 55
value, 52
base-line 14, 48
block 135
cache manager 150
Cactis 93
change layer 149
check-in 16
check-out 16
class 36, 52
Class, 55
Computed-Relation, 58
creation, 64
instance-of, 56
name, 55, 65
Piece, 115
Relation, 58
System-Relation, 58
CMS 16
command shell 146
communication 45
commutation
global, 133, 135, 142
local, 133, 140, 142
component 3, 112
conditional compilation 28
COSMOS 41
cpio 111
Damokles 22
DaSC 111
data-base
object-oriented, 51
DCWSS 1
delta 16, 32, 40
forward and backward, 16
non-text files, 32
delta-compression 16
delta-list 80
dependency set 100
derivation
graph, 89
side-effects, 99
step, 89
derivation management 36, 89
cache, 105
derivation manager 9, 17
derived value 89
equivalence, 104
DSEE 20, 28–29, 31–32, 43, 93, 95–96, 129
eager evaluation 105
Emerald 22, 38
environment 35, 41, 48, 66
Epoch 112
Epos 24, 28
frontier set 141
Gemstone 22
history
linear, 74
Index

non-linear, 76

lmake 93, 96, 98
immutability 41
inheritance
  multiple, 55
integral version management 39
Iris 22

KBSA 24, 27

language binding 62
layer 111
lazy evaluation 105
locking 14, 16, 49
long-field 150
  identifier, 150
  manager, 147
loosely-coupled distributed networks 1

Make 17, 31, 93–94, 96, 98
merging 129
  operation-based, 132
method 36, 52, 58
  functional, 58
  procedural, 58
module 3

NFS 148
NSE 19, 25, 29–31, 93, 130

Object 55
  composite, 37, 112
  creation, 63
  storing versions of, 71, 76
object management system 22
object-identifier 52
  uniqueness, 39
Odin 17, 31, 93–94, 96, 98
OMS 35, 51
Orion 22–23, 112

PACT 23, 131
Patch 18, 130
PCTE 22–23, 27, 36, 43, 112

PIE 111
  piece 48, 103, 109
    external relationships, 114
    transplant, 112, 116
  piece editor 109, 119
    rules, 122
    switches, 120
Postgres 22–23
  product 3
query 54, 65

RCS 16, 27, 29, 129
recipe 36, 58, 115
relation 36, 53, 57, 70
  Class-Name, 65
    depends-on, 101, 103, 118
    homogeneous, 57
    instance-of, 60, 116
    internal-relationships, 116
    method-of, 59–60
    name, 65
    non-homogeneous, 57
    operation, 64
    part-of, 115–116
    recipe-of, 58
    Relation-Name, 65
    storing version of, 73
    template, 57
    trigger, 61
    super-class, 55
relationship 57
release 15
rule-set 120

SCCS 16, 27, 29, 95
set
  storing versions of, 75, 80
    delta-list, 80
    trees, 82
Shape 93
shared data 45
Smalltalk 63
snapshot 35–36, 39, 66
history, 39
immutability, 39
implementation, 69
Snapshot OMS 35, 41
sparse index 74
storage manager process 147
sub-class 55
successor relation 16, 42, 71
super-class 55
tar 111
template 55
time-machine 42
time-stamp 42, 70
tool 3
transformation 42, 48, 67, 132
primitive, 132, 135
Translucent File System 111
trigger 36, 61
type 56
attribute, 56
name, 56
type evolution 23
undo 42
variant 7
management, 43
version 1
version history
linear, 70
non-linear, 70
version management 1
version-list 75
VMCM 24, 26–27, 43
weave 140
Workshop System 24, 26
workspace 3, 41
zoo 111