A Configuration Management Approach for Supporting Cooperative Information System Development

Rudolf Andersen

Information Systems Group
Faculty of Electrical Engineering and Computer Science
The Norwegian Institute of Technology
University of Trondheim
Abstract

The competitive strength of organisations is to a great extent determined by the quality of their information systems. In order to develop complex information systems on short notice, at any location, in any variety, and at the right price, it is necessary to rely on the coordinated efforts of end users and system developers working concurrently in cross functional teams. This is not possible with contemporary CASE tool technology. This thesis provides central components of a cooperative information system development environment.

We propose to expand contemporary CASE environments with functionality which:

- **allows for controlled chaos** in true sharing of the system specifications among the members of a development team. The same development object may be modified by several members of the team, thus creating potential update conflicts in the specification database. Modifications are performed in the context of development transactions.

- **monitors potential conflicts** by keeping track of the various local versions that are being created by different members of the development team. Publish and subscribe mechanisms provide for exchange of local versions among development transactions. As long as potential conflicts exist, no development transaction is allowed to update the specification database.

- **supports conflict resolution through negotiations** by integration of groupware technology, e.g., joint editing, and multimedia technology, e.g., video conferencing and voice annotations. Negotiations may be either real-time or asynchronous.

- **supports versions of specifications where specificational detail have been removed** through a process of abstraction. It is necessary to abstract away details so that simplified views of the system may be used for communication purposes. Modifications may subsequently be made on the abstracted view instead of on the fully detailed view.

A framework for management of change in a cooperative information system development environment which includes the functional components listed above is proposed. A design is proposed to integrate this framework with the PPP ICASE environment.

A prototype of the framework customised to PPP requirements, but so far restricted to textual objects, has been implemented. The prototype utilises an extended RDBMS to implement a specification repository for PPP.
Preface

This thesis is submitted to the Norwegian Institute of Technology for the doctoral degree "doktor ingeniør". The work reported has been carried out at the Information Systems Group, Faculty of Electrical Engineering and Computer Science, the Norwegian Institute of Technology, The University of Trondheim, Norway, under the supervision of Professor Arne Sølvberg. I have also stayed at the Department of Information Processing and Computer Science, the Royal Institute of Technology and the Swedish Institute for Systems Development (SISU), Stockholm, Sweden from September 1989 to February 1990.

During the years of my study I was employed as a research assistant (1985 - 1987), and as a lecturer (1988 - present) at the Faculty of Electrical Engineering and Computer Science, the Norwegian Institute of Technology, Norway.

Acknowledgements

This thesis has been influenced by a number of people. In particular, I would like to thank the following people:

- Professor Arne Sølvberg for continuous encouragement and criticism, and for never failing to introduce me to challenging tasks that put the thesis work into perspective.
- My colleagues in the Information Systems Group for providing an inspiring working environment.
- The students I have supervised, particularly Stefan Weiblen, Gerald Kimmel, and Anders Bjørnestad for their implementation efforts and for their constructive criticism of my ideas.
- Professor Janis A. Bubenko jr. for giving me the opportunity to stay six months at his research institute in Stockholm, Sweden.
- My dear wife Linda for her patience.

Trondheim, September 1994

Rudolf Andersen
# Contents

1. Introduction .................................................................................................................. 11
   1.1. The problem .......................................................................................................... 12
   1.2. Concurrent cross functional teams ................................................................. 13
   1.3. Weaknesses in contemporary CASE .............................................................. 14
   1.4. Our approach ...................................................................................................... 15
   1.5. Major achievements .......................................................................................... 16
   1.6. The structure of the thesis ................................................................................. 17

2. The CASE tool domain ............................................................................................... 19
   2.1. Information systems engineering ................................................................. 19
       2.1.1. Systems life cycle .................................................................................. 19
       2.1.2. System models ..................................................................................... 21
   2.2. Characterisation of CASE tools .................................................................... 23
       2.2.1. Definition of CASE ........................................................................... 23
       2.2.2. What is and what is not CASE .......................................................... 24
       2.2.3. Features of CASE .............................................................................. 26
   2.3. Analyst/Designer’s workbenches ................................................................... 31
   2.4. Code generators ............................................................................................... 34
   2.5. Life cycle tools .................................................................................................. 36
   2.6. Repositories ...................................................................................................... 38
   2.7. Trends in CASE tool development .................................................................. 40
   2.8. Chapter summary .............................................................................................. 42

3. Cooperative work support ......................................................................................... 45
   3.1. Aspects of groupware ....................................................................................... 45
   3.2. Taxonomies of groupware ............................................................................... 48
   3.3. Categories of groupware .................................................................................. 49
   3.4. Chapter summary .............................................................................................. 55

4. Version and configuration management .................................................................... 57
   4.1. Version and configuration management features ........................................... 57
       4.1.1. Definition of version and configuration management .......................... 57
       4.1.2. Categories of version and configuration management approaches .......... 58
       4.1.3. Dimensions of version and configuration management ...................... 59
   4.2. File based systems ............................................................................................. 63
       4.2.1. SCCS - The Source Code Control System ........................................... 63
       4.2.2. RCS - The Revision Control System ....................................................... 65
       4.2.3. Make - A Program for Maintaining Computer Programs ................. 66
       4.2.4. DSEE - The Domain Software Engineering Environment ............. 68
       4.2.5. Huff’s Model ......................................................................................... 69
       4.2.6. Lockeman’s Model ............................................................................... 71
       4.2.7. Tichy’s AND/OR Model .................................................................... 74
   4.3. SCM for engineering databases ........................................................................ 78
       4.3.1. Damokles ............................................................................................. 78
       4.3.2. Change-Oriented Versioning ................................................................. 79
       4.3.3. Farandole 2 .......................................................................................... 83
   4.4. Support for SCM in commercially available CASE tools ............................... 86
       4.4.1. SCM in Foundation ............................................................................ 86
       4.4.2. SCM in Information Engineering Facility ............................................ 87
       4.4.3. SCM in Software through Pictures ....................................................... 88
   4.5. Chapter summary .............................................................................................. 89

5. Management of change .............................................................................................. 91
5.1. Version and configuration management ............................................. 91
  5.1.1. Development object versions, an example .................................. 91
  5.1.2. Definition of development object versions ............................... 93
  5.1.3. Definition of component structure ........................................ 95
  5.1.4. Global name uniqueness .................................................... 110
  5.1.5. Granularity of versioned components ..................................... 111
  5.1.6. Selecting a version .................................................................. 112
5.2. Way of working ............................................................................. 113
5.3. Development transactions ............................................................. 116
  5.3.1. Properties of development transactions .................................. 116
  5.3.2. Check-out and check-in ......................................................... 117
5.4. Development teams ....................................................................... 119
  5.4.1. Synchronising transactions .................................................... 120
  5.4.2. Conflict resolution .................................................................. 123
5.5. Chapter summary ........................................................................... 127

6. Change support for complex models ................................................ 129
  6.1. Abstractions ................................................................................. 129
  6.2. Properties of abstractions ............................................................ 136
  6.3. Modifications of abstractions ....................................................... 141
  6.4. Conceptual model of development transactions ............................ 149
  6.5. Chapter summary ....................................................................... 149

7. Management of change for a CASE tool repository .......................... 151
  7.1. Introduction to PPP ...................................................................... 151
  7.2. Requirements to a repository for PPP .......................................... 157
  7.3. Implementing cooperative work management functionality ............ 160
    7.3.1. Alternative integration strategies ........................................ 160
    7.3.2. Architecture of the chosen solution .................................... 163
  7.4. Repository schema design ........................................................... 165
    7.4.1. Objects from several domains ............................................. 165
    7.4.2. Choosing development object granularity ............................ 167
    7.4.3. The granularity of the PrM and PhM .................................. 170
    7.4.4. Alternative storage solutions ............................................. 172
  7.5. Chapter summary ....................................................................... 176

8. Conclusions and future work ............................................................ 177
  8.1. Main conclusions ........................................................................ 177
  8.2. Claimed contributions .................................................................. 177
  8.3. Directions for future work ............................................................ 178

Appendices .......................................................................................... 179

A. IEF an example of an ICASE tool ...................................................... 181
  A.1. The single user toolsets ............................................................. 182
  A.2. The central encyclopedia tools .................................................... 184

B. IPSEs - Integrated Project Support Environments ............................ 189
  B.1. Important IPSE issues ................................................................. 191
  B.2. Towards the information systems factory .................................... 193

C. PCTE - Portable Common Tool Environment .................................. 197
  C.1. The object management system ................................................ 197
  C.2. The process model ..................................................................... 199
  C.3. The user interface facilities ....................................................... 200
  C.4. PCTE as an open repository ...................................................... 201

D. IBM's AD/Cycle and repository manager ........................................ 203
Contents

D.1. AD/Cycle architecture .................................................. 203
D.2. Cross life cycle activities .............................................. 204
D.3. AD/Cycle platform architecture ..................................... 205
D.4. The repository manager ................................................ 206
D.5. The information model .................................................. 207

E. A meta model for PPP .................................................. 211
   E.1. Static modelling constructs, PhM ................................ 211
   E.2. Dynamic modelling constructs, PrM ............................. 214

F. The PPP storage structure ............................................ 217
   F.1. Fact for overall system structure ................................ 217
   F.2. Facts for PhM diagrams .......................................... 217
   F.3. Facts for PrM diagrams .......................................... 219

G. PPP facts of a detailed example ..................................... 225

H. Object structure of a detailed example ............................ 229

I. The repository schema for the ERDBMS alternative ............. 243

References ................................................................. 247
1. Introduction

The most important power source of our generation is information processing. The development in computer hardware technology has been spectacular. If we look back we may observe more than six orders of magnitude price-performance gain over the last 30 years. This is true for CPU power, primary storage, secondary storage, and communication bandwidth [22]. This situation holds much promise for the future.

The massive computing power which has been brought to our fingertips will transform the world’s industries and the people who work in them. The centerpiece of the business revolution will be a new kind of product. These products can only be built now due to the latest innovations in information processing, organisational dynamics, and manufacturing systems. But more importantly, the products can be made available at any time, in any place, in any variety, and at the right price. They are called virtual products [34].

Virtual products exist before they are produced. Their concept, design, and manufacture are stored in the minds of cooperating teams, in computers, and in flexible production lines. For example, the Japanese are striving to “virtualise” the production of cars by putting in place systems that will produce cars to domestic order in just seventy-two hours [35].

The challenge that these virtual products’ production lines put on information system support is massive. However, the software community has not been able to keep pace with the demand for new and increasingly sophisticated software. The software crisis was identified already in the 60’ies. The term software engineering was coined in 1968 as an approach to counter the crisis. However, software crisis may still be used as a characterisation of the state of affairs within the software community.

One of the reasons why software productivity has not followed the same pattern of growth as hardware technology may be that while hardware production has followed a standard capital intensive engineering approach, software production is still very much a labour intensive activity.

Another observation is that contemporary software engineering approaches that have been practised since 1968 have managed to solve virtually all of the “simple” problems. The information systems that remain to be developed are all intrinsically complex and of significant size. Developing this kind of systems often require that knowledge and skills from many problem domains and areas of expertise are brought together and allowed to crossfertilise. The markets pull for more integration of systems that were previously standalone increases the complexity of the system development tasks even more. Hence, the magnitude and complexity of the problems we are facing today are such that they can only be solved within the context of crossfunctional teams.

During the last 10 years we have seen the rise of a new field in system development. Computer aided software engineering (CASE) aims to make software production into an engineering discipline supported by efficient and effective tools. Many models for system development have been proposed over the last 20 years. They are increasingly being supported by CASE tools. In fact, no proposed model can hope for success unless sup-
ported by a CASE tool. Still, contemporary CASE tools are strong on supporting the work of individual developers and weak on supporting the communication and coordination amongst a team of developers.

1.1. The problem

The primary goal of anyone who shall survive in the area of information system production in the future is to be able to deliver the information systems at any time, in any place, in any variety, and at the right price.

To achieve this goal we identify three important areas which need to be supported: end user involvement, cooperation in cross functional teams, and the ability to work concurrently.

Development and maintenance of computerised information systems are characterised by continuous modifications to the specifications. Modifications are an integral part of the process of stabilising requirements, designing alternative solutions, implementing a chosen design, maintaining an operational system, etc. They are motivated by a mismatch between the requirements to the system and the current state of the systems specifications with its implications on actual system behaviour. A mismatch leads to increased costs and reduced benefits when using the system. Hence, modifications are made to improve the cost/benefit ratio associated with using the system. The cost/benefit ratio can only be determined by the end user so the end user must be involved in the modification processes. The dialogue with the end users must be in terms of concepts that are familiar to the end user. Hence, the focus will be on information system specifications with conceptual and functional focus and not on code.

Development and maintenance of information systems of some size require significant efforts. The systems are usually large, complex, and unsurveyable [93][140]. Modifications of unsurveyable systems with high productivity usually require coordinated efforts of several people. These people will exhibit diverse backgrounds and profiles of competence and represent different functions in the organisation. Coordination covers aspects of organisation and the management of resources (people and systems), control of the use of resources, and the provision of and oiling of the interaction links that are required [106]. Thus, the "art" of providing support for cross functional teams becomes an important skill in tomorrows system development environments.

If the information system products are to be delivered at any time, usually on short notice, not only is it necessary to provide for cross functional competence, but it is also necessary to support the concurrent work of the members of the cross functional teams. The size of the information systems and the multitude of the modification sub-tasks will usually be so large that a serial ordering of the tasks will prove infeasible and make it impossible to meet the required delivery time. Hence, the only feasible way to deliver on time, is to work concurrently.

To achieve these three key elements, i.e. end user involvement, cooperation in cross functional teams, and the ability to work concurrently, one possible approach is to deploy many short modification cycles, to maintain an information system product model which may be dynamically
changed, and to facilitate parallel (or concurrent) use of the information system product model.

To be able to do this under good control it is necessary to support multiple "ownership" of versions of the information system product model both on an individual and a group basis. Furthermore it is necessary to be able to handle the conflicts that arise from concurrent work.

1.2. Concurrent cross functional teams

When an information system is modified a new version of the system is created. The modification process is called a development transaction in the sequel. Several versions of an information system may be developed simultaneously, and the dependencies between the versions must be kept. A new version may exist in parallel with or may replace a previous version.

We distinguish three ways of working in a development team: solitary work, cooperative work, and coordination [46].

**Solitary work** is performed by a single person. It focuses on production and thinking. Typically the individual will work on a limited part of the information system specification. The work is done at interleaved periods of time, and is thus asynchronous in nature.

**Cooperative work** is also oriented towards the product. The difference between solitary work and cooperative work is that in the latter work mode a group of two or more are collaborating on solving a task. They are developing their joint work while communicating, e.g., exchanging ideas and viewpoints, and evaluating and negotiating to find the best solution. The communication may be synchronous or asynchronous. In addition to writing and drawing, group members talk, make gestures, present pictures, etc. Group members may be located at the same or different sites.

**Coordination** is performed by a group. The focus is on the process and not the product. The setting is typically a meeting where progress and plans are discussed. Planning, evaluation, and decisionmaking are central tasks. The way of working is usually synchronous and "bursty".

Development teams require facilities for sharing system specifications among the members of the team. Following our ways of working the facilities must support true sharing of specification while maintaining support for cooperative work and point of coordination. The facilities must allow the developers to modify the system simultaneously. If simultaneous modifications of the same system component are permitted points of coordination among developers must be supported. Support for merging individual developers' modifications into a new consistent version of the system may also be necessary.

The specifications of non-trivial systems are complex and contain many details. It is impossible for human beings to perceive large numbers of detailed specifications unless the specifications are considerably simplified, for example by hiding details in the specifications. Specifications where details have been removed are called abstractions. It is necessary to sup-
port system development, including modification of systems, by providing abstractions so that simplified views of the system are used for communication purposes. The modification will then be made on the abstracted view instead of on the fully detailed view.

When the modification process has reached a stage where one is about to add details that go beyond what a particular abstraction supports, one may continue the modification process on specifications that comprise a full set of details. To return from the abstracted specifications to fully detailed specifications one may reintroduce details that were previously hidden in the abstracted view. More details may then be added to complete the modification in specifications with full details. A consequence of this is that abstractions have to be supported, e.g., by version control, in much the same way as the modifications of the fully detailed specifications must be supported.

1.3. Weaknesses in contemporary CASE

Contemporary CASE (Computer Aided Software Engineering) technology offers limited support for people who work in teams. The standard CASE tool consists of analysis and design workbenches, code generators and other standalone tools. The primary focus of ICASE tools is on integration across development stages. For that reason the tools relate to a specification repository. The existence of a specification repository makes sharing of specifications possible among tools. Facilities for the sharing of specifications are nevertheless still limited. This makes it difficult to use them in a team context.

Specification repositories are also called encyclopaedias, dictionaries, or simply specification databases. The repositories are either offered as standalone systems, e.g., Repository Manager from IBM [73], or as the hub of an integrated system development environment, e.g., the Central Encyclopedia in Information Engineering Facility, IEF [142].

The team interaction patterns that are offered by contemporary CASE tools are restricted. Contemporary repositories are based on the assumption that it is possible to undertake system development in teams by splitting the system specification into disjoint subsets. Each subset is to be further developed by individual team members. For example, in IEF [141][142], subsets of the specification are checked out during each "development transaction". The checked out subsets of the specification are marked in the repository as being locked. The revised specifications are checked back into the repository when the development transaction is over. During a development transaction the development objects in the repository that are locked cannot be modified by others.

Arguments and empirical evidence seems to indicate that independence between development objects is extremely difficult if not impossible to achieve [92][32]. This may of course be so because one has not been sufficiently clever in choosing disjoint subsets. However, it seems to be very difficult to determine useful disjoint subsets unless a detailed analysis has been performed. One simply can not know what possible dependencies exist among the development objects without doing a detailed analysis. To be able to do this detailed analysis it is necessary to split the work and consequently to split the development objects into disjoint subsets, due to the size
and complexity of the typical problem. This is the classical chicken and egg situation.

1.4. Our approach

We propose that contemporary CASE technology is expanded with functionality for improving system development performed by teams of individual developers:

- Allow for controlled chaos

  We permit a distributed development situation where the system specifications are shared among the developers. A relaxed set of authorisation rights is proposed, such that objects may be checked out to many simultaneous development transactions, thus creating potential update conflicts. It may well be that several transactions modify the same object. Several transactions may all try to update the same part of the repository.

- Monitor potential conflicts among developers

  The different local system versions of the different development transactions must be monitored such that potential conflicts can be detected and managed. Publish and subscribe facilities provide for exchange of local object versions among transactions. As long as there is a potential conflict, no transaction is allowed to update the repository.

- Help resolve conflicts through negotiations

  An update conflict will have to be resolved in one of two ways: (1) The developer responsible for the object in question unilaterally dictates a solution to the other interested parties. (2) A negotiation process is initiated among the transactions with potentially conflicting local versions of the same object. The negotiation process should be supported by groupware technology, e.g., joint editing functions, and multimedia technology, e.g., video conferencing and voice mail.

A successful implementation of the proposed functionality for conflict management depends on resolving the two issues of: (1) Integration with a shared repository where the public specifications may be kept. This repository must offer a versioned interface to the system specifications. (2) Integration with a feasible CASE tool.

A specification repository contains all relevant development project data to be shared by the interested parties, e.g., system analysts, system designers, programmers, and project managers.

To satisfy the requirements from this heterogeneous set of users, the repository must encompass concepts from four different domains: The subject domain, the implementation domain, the development object administration domain, and the development project domain.

Feasible CASE tools record and maintain data from the subject domain and the implementation domain. Conflict management integrated with adequate version and configuration management will manipulate data from the development object administration domain. Functionality necessary to manipulate concepts in the development project domain may be
added. The project management aspects will not be considered in this thesis.

The specification repository will thus contain interrelated sets of data that are partly maintained by the CASE tool, partly by conflict management functionality, partly by version and configuration control, and partly by project administration software. Precisely which concepts that are used and what the relationships between them are, are determined by the meta models of the CASE tool, the software for conflict management, version and configuration control, and project administration respectively. These meta models must therefore be externalised.

In summary, what we would like to achieve in this thesis is to develop an environment for supporting cooperative information system development which:

- reflects on the current weaknesses in cooperative information system development.
- consists of a version control based approach to coordination of system development in teams.
- is designed such that it may be integrated in a CASE technology framework.

The discussion above indicates that to improve the support for team work in contemporary CASE tools, solutions must be found in the following areas:

- Version and configuration management
- Cooperative approaches and conflict management
- Integration with a feasible CASE tool

From this we determine the need to establish the state of the art and further our understanding within these three fields: the CASE tool field, the computer supported cooperative work (CSCW) field, and the version and configuration management field. Based on the background knowledge acquired through this state of the art we aim to build our research contribution.

1.5. Major achievements

The outcome of the work is a framework for cooperative information system development which is based on true sharing of development objects and management of conflicts. The framework is applied to provide a multi user repository for the experimental ICASE environment PPP.

The main contributions of the thesis are:

- A more flexible approach to cooperative information systems development.
  The approach permits sharing of development objects, and resolution of update conflicts through negotiations. The approach provides improved integration of synchronous and asynchronous modes of working.
- Integration and utilisation of feasible groupware components.
The improved integration of synchronous and asynchronous modes of working is due to the integration of version and configuration management, management of development transactions, and negotiation support through groupware components, i.e. joint editing functionality, video conferencing, multi-media annotation and mail functionality.

- Augmenting the functionality of CASE repositories.

A framework that permits cooperative information system development have been proposed. It has been shown that the approach may be integrated into existing CASE tools, provided the meta model implemented by the tool is externalised.

1.6. The structure of the thesis

The structure of the thesis is as follows:

Chapter 1 is this introduction.

Chapters 2, 3, and 4 respectively, contain a state of the art survey on CASE tools, on support for cooperative work, and on version and configuration management approaches.

Chapter 5 contains our proposal for management of change in a cooperative information system development environment. It comprises a version and configuration management scheme, a way of working, a scheme for development transactions, and a scheme for supporting development teams including a conflict resolution mechanism.

In chapter 6 the proposed change management framework is applied to complex models. The framework is augmented with functionality necessary to manage the process of simplification of the models by abstracting away specificalional detail.

In chapter 7 we investigate the integration of the change management framework with the experimental ICASE environment PPP. This comprises a description of the PPP, requirements to a repository for PPP, a discussion on alternative integration architectures, and a design of a repository solution for the PPP tool.

Chapter 8 contains conclusions, claimed contributions, and future work.

The appendices A, B, C, and D respectively, contain detailed descriptions of the ICASE tool IEF, the IPSE tool category, the PCTE object store, and the AD/Cycle application development framework.

The appendices E and F contain a meta model for PPP and the PPP storage structure.

The appendices G and H contain the PPP facts of a detailed example and the corresponding object structure for the example.

Finally, appendix I contains the repository schema for an extended RDBMS implementation of the PPP repository.
2. The CASE tool domain

There are more than 100 commercial tools on the market today that fall into the CASE tool category. Several market surveys are available, e.g., [64][65][120][121]. Several scientific and commercial conferences aim at shedding some light onto this field [115][134][7][104][122].

This chapter gives a state of the art survey of the CASE tool domain. It comprises an overview of the methodological foundation of information system development environments, followed by a definition of the CASE tool domain by analysing characteristic features, a survey of analyst/designer's workbenches, code generators, life cycle tools, and repositories. Selected CASE approaches have been described in detail in appendices A, B, C, and D. The chapter closes with a discussion of trends in CASE tool development.

2.1. Information systems engineering

This section gives an overview of important aspects of the methodological foundation of contemporary information system development environments.

2.1.1. Systems life cycle

Information systems engineering is the application of rigorous engineering principles to the process of designing and constructing an information system. The reason for building information systems is to provide correct and timely information to technical and/or administrative organisations. The objective of the engineering process is to find a satisfactory compromise between the quality of the information system and the effort spent in developing it.

The systems engineering process

The engineering process is characterised by a number of stages. Jensen and Tonies [81] suggest six engineering development stages. The names and numbers of stages found in the literature might differ, but essentially they are the same.

- **Problem formulation** A definition or description of the problem in broad terms without detail is proposed.
- **Problem analysis** The problem definition is redefined to supply essential detail.
- **Solution generation** A set of potential solutions to the problem is developed.
- **Solution selection** The alternative solutions are evaluated and compared, and the best solution is chosen.
- **Design specification** The chosen solution is described in detail.
Implementation

The finished product is constructed from the design, tested, and installed.

To these six stages one might add a seventh stage:

Modification

The system is modified as time passes to satisfy new requirements.

It is debatable whether modification should be regarded as a separate development stage because modification encompasses all of the six previous stages. Each time that a design is modified one has to proceed from problem definition to implementation, in order to treat modification proposals in an orderly way.

There are three major classes of modifications:

- **Corrective modifications**, for repairing the system to change unacceptable systems behaviour.
- **Adaptive modifications**, to change system functionality because of environmental changes, customer wishes, competitors' pressure, etc..
- **Perfective modifications**, to change the system quality with respect to operational cost, modifiability, etc..

Each modification that is applied to a system essentially run through all the first six stages, be it a corrective, adaptive, or perfective modification. Depending on the nature of the modification, some stages may be run through rather briefly and some stages will be performed more rigourously. The objective of the modification process is to improve an existing system in some way. Thus, a seemingly minor modification may have large and unsurveyable side effects [13]. Thus the need for tool support during the modification process seems evident.

The system life cycle

The sequence of development stages where a system is developed from early ideas and imprecise requirements into an operational information system is called the life cycle of the information system. The life cycle is frequently called the waterfall model of system development, implying that the results from one development stage falls into the next stage thus feeding the corresponding development activity, the results of which fall into the next stage, and so on.

Viewing system development in such a way implies that there is a clear distinction between what to do and how to do it. In other words, it must be possible to separate the problem and its solution to such an extent that a complete problem formulation can be worked out prior to the solution of the problem. When the engineering development process is described as a sequence of stages, this indicates that each stage may be more or less completed before moving on to the next stage. This would be a very attractive situation from the point of view of managing the engineering project. We could then make a final evaluation of the results of each development stage prior to moving on to the next stage, and we would never have to look back.
Unfortunately the world is not that simple. At each development stage new insights into previous decision situations become available. These insights may modify or invalidate the decisions that have led us to our present understanding of the problem, thus forcing modifications to previous development stages.

Some central concepts are explained in the following:

**Development stage** is the elementary component in the systems life cycle. The development stage $i$, figure 2-1, contains a formulation of a solution, the **how**$(i)$ that satisfy a requirement, the **what**$(i)$. The requirement statement **what**$(i)$ equals the solution **how**$(i-1)$ of the previous development stage. Thus, every stage in the engineering development life cycle is called a development stage.

![Fig. 2-1: Development stages](image)

**Development objects** are inputs and outputs of development stages. The development objects developed during development stage $i$ act as input to development stage $i+1$. The term **system specification** usually means the collection of relevant development objects. The term **system component** will also be used as a synonym to the term **development object**.

### 2.1.2. System models

Information system models reflect on the real world system. Such models are of two classes: constructive subsystem structures and non-constructive subsystem structures.

**Constructive subsystem structures**

Successful application of the engineering development principles reported in the previous section require that the conceptual models make it possible
to verify that a solution proposed in step $i+1$ satisfy the requirements formulated in step $i$. This property is termed constructivity [93]. The following is a definition of constructive subsystem structures as originally proposed by Langebors and modified by Sølvberg and Kung [140].

"A constructive subsystem structure of a system is such that the properties of the system can be derived (constructed) when the relevant properties of the system's components are known.

An applicable subsystem structure of a system is such that the properties of the system components together with the correlations among the components result in the properties that have been specified for the system as a whole.

An implementable subsystem structure of a system is such that every system component and correlation can be implemented."

This can be restated in the following way:

"The only possibility of designing a system to have specified properties, is to design an applicable and implementable subsystem structure for that system."

Models with constructive properties will exhibit a number of features:

**Formal language** It must be possible to state requirements and proposed solutions in a language that allow automated checking of the fulfilment of the requirements. This translates into a formal logic based language.

**Uniform language** The language used during the various stages will be the same in the sense that the constructs provided in early stages will be a subset of the constructs provided in later stages.

**Executable specifications** The language must contain sufficient computational semantics to allow execution of the specifications at every level of detail.

**Incremental specifications** Following from the uniform language, the objects developed during stage $i+1$ will be based on the objects developed during stage $i$, in such a way that the specifications of stage $i+1$ augment the specifications of stage $i$.

**Non-constructive systems**

Models without the constructivity property will not exhibit all of these features. The objects developed during the various stages may not relate properly to each other. E.g., many of the modelling concepts used for requirements capture may not be used or may not have any meaningful interpretation relative to the concepts used for developing design specifications. Languages used for requirements capture, technical design, and implementation are usually different from each other, thus making it very difficult to pass development objects between development stages. Verification and validation of specifications can consequently only be done by inspection or by implementing (part of) the technical design. It is usually not possible to automatically verify that a technical solution satisfy the requirements to the system.
It is thus impossible to ascertain that objects developed in stage \( i+1 \) will satisfy the requirements formulated by the objects developed in stage \( i \), except by manual inspection.

E.g., consider the Structured Analysis and Design (SA/SD) approach [55]. Techniques used in the SA/SD include the well known data flow diagrams (DFD) and structure diagrams (SD). Some form of entity-relationship diagrams [25] are also often used in the SA/SD approach. The problem analysis, solution generation, and selection stages are documented using the DFDs. The selected solution is developed into a technical design which is expressed in SDs. With the exception of "transform centered design" approach [55], there are few rules or guide-lines for developing a technical design from a DFD. The use of a new language to document the technical design increase the difficulties. There are no provisions for relating the development objects in the technical design (expressed in SDs) to the development objects in the functional solution (expressed in DFDs).

2.2. Characterisation of CASE tools

This section gives a characterisation of the CASE tool domain from two different points of view; on the one hand we discuss definitions of CASE, and on the other hand we discuss different dimensions which may be used to characterise CASE tools and thus improve our understanding of the domain.

2.2.1. Definition of CASE

A key issue is the definition of the term itself. A very general definition is provided by Forte and Norman [51]:

"Tools and methods to support an engineering approach to software development at all stages of the process. By engineering approach we mean a well-defined, coordinated and repeatable activity with widely accepted representations, design rules and standards of quality. Tools that support such a software engineering discipline are, by our definition, CASE tools, regardless of the specific phase, task or notation."

This definition is very broad spectred. It covers tools that are based on diagrammatic specifications, e.g., analyst/designers workbenches, ICASE tools, and repositories. It also includes tools based on textual specifications, e.g., (structure) editors, compilers, and debuggers. The latter tools emerged long before CASE became a topic. This definition is too broad for our purposes. We focus on tools for diagrammatic specifications and integrated toolsets which starts off with diagrammatic specifications and offer support all through to executable code.

A more appropriate definition for our purposes may read as follows:

"CASE tools are computer based tools to support all stages of information systems engineering from diagrammatic specifications to executable code."
We have chosen to omit an explicit mention of methods from our definition because tools are implemented to support specific system development methods and not vice versa. The ideal situation is that having decided on a tool one has also decided on a method. Having said this, we are painfully aware of that many contemporary tools have weak or non-existent method support and as such operate more as drawing editors rather than CASE tools. We believe that this defect is largely due to the infancy of the tools, and it is constantly being improved.

Diagrammatic specifications are most commonly used in the early stages of the systems life cycle. This puts our focus on tools that support early stages of the life cycle and rules out programming oriented tools.

By our definition, it is an important property of a CASE tool to be able to produce executable code from diagrammatic specifications. This may be realised on several levels of ambition. The low end of the spectrum is seen in the structured language of Action diagrams in IEF[141]. The high end is represented by diagrams which record sufficient computational semantics to allow execution of the diagrams, a functionality rarely seen in commercial tools. A notable exception is the SDL language used for real-time system modelling and design [24][14] and the tools supporting the language [53][54].

2.2.2. What is and what is not CASE

Even within our definition of CASE, the number of tools on the market is still large. Some of the tools are CASE tools according to our definition, some are in the "grey" area, and others are not CASE tools. The names of the various tools have been influenced by the functions performed by the tool. E.g., there are analyst/designers workbenches which support structured analysis and design, code generators which generate executable code mostly from textual specifications, and ICASE tools which support development of operational systems through integrating specifications all the way from diagrams down to executable code.

Some of these tools, e.g., some of the code generators, date back to the 70's and have only recently been redefined as CASE tools. While others, e.g., the analyst/designers workbenches, were introduced during the 80's and were the tools that actually started the CASE tool as a business area.

A classification on what is and what is not CASE has been proposed by Hewett and Durham [65], and is depicted in figure 2-2. The discussion of the CASE tool domain in this thesis will only consider a selection of tools from the two innermost layers of the figure: Analyst/designers workbenches, code generators, life cycle tools, and repositories.

**Analyst/designers workbenches** typically support structured analysis and design [55]. There is a strong focus on manipulation of diagrammatic specifications. The tools often provide support for several specification models, e.g., data flow diagrams, entity relationship diagrams, structure charts. There is weak integration between the models, e.g., there is generally weak consistency checking between data flow diagrams and entity relationship diagrams. The dictionary support is weak. The tools give little or no help in going from analysis to design e.g., it is generally not possible to have a structure chart
generated from a data flow diagram. Finally, there is little support for transforming the designs into executable code.

**Code generators** generate executable code from mostly textual specifications in very high level programming languages, screen paintings, fill-in screen forms or menus. The tools are usually not related to analyst/designers workbenches. There is strong emphasis on support for code manipulation, code generation, and prototyping. The tools offer advanced support for maintenance related activities such as testing, cross-referencing, and version control.

![Diagram](image)

**Fig. 2-2: What is a CASE tool? [65]**

**Life cycle tools** focus on giving support all the way from a diagrammatically oriented systems analysis stage through to generation of code. The generated code range from prototypes of various kinds, via skeletal structures that need to be completed manually, e.g., StP [76], to fully operational code, e.g., IEF [141]. The tools put strong emphasis on integration of different models. There are two dimensions to this integration; “horizontal” integration among models used during the same life cycle stage, e.g., consistency checking between data flow diagrams and entity relationship diagrams, and “vertical” integration among models used during different life cycle stages, e.g., consistency checking between data flow diagrams and structure charts. There is also strong emphasis on providing support all the way through to executable code, i.e. the job is not done until the result is an operational application. The approach is strongly repository based with the ambition of maintaining a single canonical representation of the whole application where the different models used during the life cycle stages may be considered as logical views of this representation [51]. The tools often provide project management support and support for office functions. Some of the life cycle tools are open ended in that they support integration of tools through the frame-
work provided by the representation of specifications in the repository.

Repositories are specification databases for other CASE tools, e.g., analyst/designers workbenches and code generators. There is strong emphasis on providing a generic extensible framework for integration of tools. The integration is based on integration of development concepts from the different tools. There is also focus on and management of development objects from the individual tools.

2.2.3. Features of CASE

The benefits of a concise definition of CASE may be outweighed by its inability to reflect on a broader set of characteristic properties to further define the term. This section tries to remedy this by offering a number of dimensions to further characterise our selection of CASE tools and thus improve our understanding of the domain.

We propose the following dimensions to analyse CASE tools: The functionality supported, the focus of the tool on a specific (set of) life cycle stage(s), the integration across life cycle stages, the degree of method support, the “closed” (proprietary) or “open” (published/standard) specification database formats, support for teams of developers, hardware/software platform, and market position.

The functionality supported

Functionality may be classified into four (partly overlapping) categories:

- Functionality for subject domain modelling (S) is concerned with the problem domain, and with building a model of the real world that caters for behavioural, performance, and environment interaction aspects. It is important that the model lends itself to transformation into target system specifications.

<table>
<thead>
<tr>
<th>Subject domain modelling (S)</th>
<th>Analyst/design</th>
<th>Code gen.</th>
<th>Life cycle</th>
<th>Repositories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development &amp; maintenance</td>
<td>**</td>
<td>**</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>Data modelling</td>
<td>**</td>
<td>**</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>Process modelling</td>
<td>**</td>
<td>**</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>Hierarchical decomposition</td>
<td>**</td>
<td>**</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>Verification, validation &amp; testing</td>
<td>*</td>
<td>**</td>
<td>**</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2-3: Features of subject domain modelling

- Functionality for implementation domain modelling (I) is concerned with manipulating information system specifications on various levels of abstraction. The system specifications are related to both the problem domain model and the component administration model.
• Functionality for development object administration modelling (A) is concerned with management of information systems and system components. Concepts such as versions and configurations of systems and system components are related to the system specifications on one hand, and the development project organisation on the other.

<table>
<thead>
<tr>
<th>Implementation domain modelling (I)</th>
<th>Analyst/design</th>
<th>Code gen.</th>
<th>Life cycle</th>
<th>Repositories</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Development &amp; maintenance</strong></td>
<td>**</td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data modelling</td>
<td>**</td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process modelling</td>
<td>**</td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hierarchical decomposition</td>
<td>**</td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Editing</strong></td>
<td>**</td>
<td>*</td>
<td>**</td>
<td>*</td>
</tr>
<tr>
<td>Syntax checking</td>
<td>*</td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semantic checking</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3GL code checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiling and linking</td>
<td></td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Document generation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code reuse</td>
<td></td>
<td></td>
<td></td>
<td>**</td>
</tr>
<tr>
<td>Design reuse</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reverse engineering</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance optimisation</td>
<td>**</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Verification, validation &amp; testing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prototyping</td>
<td>**</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td></td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>Debugging</td>
<td></td>
<td></td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>Theorem proving</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symbolic evaluation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tool support</strong></td>
<td></td>
<td></td>
<td></td>
<td>**</td>
</tr>
<tr>
<td>Tool generation and customising</td>
<td></td>
<td></td>
<td></td>
<td>**</td>
</tr>
<tr>
<td>Tool integration</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2-4: Features of implementation domain modelling

<table>
<thead>
<tr>
<th>Development object administration modelling (A)</th>
<th>Analyst/design</th>
<th>Code gen.</th>
<th>Life cycle</th>
<th>Repositories</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object management</strong></td>
<td>**</td>
<td>**</td>
<td></td>
<td>**</td>
</tr>
<tr>
<td>Version control</td>
<td>**</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration management</td>
<td>**</td>
<td>**</td>
<td></td>
<td>**</td>
</tr>
<tr>
<td>Defect tracking</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirement tracing</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Dependency management</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code reuse</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Design reuse</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Access control</td>
<td></td>
<td></td>
<td></td>
<td>**</td>
</tr>
<tr>
<td><strong>Support</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Office automation</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

Fig. 2-5: Features of development object administration modelling
- Functionality for development project modelling (P) is concerned with monitoring and managing the development project organisation.

The contents of figures 2-3, 2-4, 2-5, and 2-6 is compiled partly from [120], [121]. For each feature chosen the figures summarises the tool category in which it is applicable and may be found in tools existing in the marketplace.

<table>
<thead>
<tr>
<th>Development project modelling (P)</th>
<th>Analyst/design</th>
<th>Code gen.</th>
<th>Life cycle</th>
<th>Repositories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project management</td>
<td></td>
<td></td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>Estimation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Planning</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling</td>
<td></td>
<td></td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>Monitoring</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2-6: Features of development project modelling

The life cycle stages

Tools may be classified into front-end tools (upper CASE), back-end tool tools (lower CASE), and life cycle tools. Front-end tools focus on the early stages of the life cycle where system analysis and design of a solution are typical activities. An example of a tool category used during these stages is the analyst/designers workbenches. Back-end tools focus on the later stages of the life cycle where detailed design and realisation of a design are typical activities. An example of a tool category used during these stages is the code generators. The life cycle tools are characterised by their support for all, or nearly all, of the development life cycle stages. The tools offer integrated support from system analysis, via design, through to realisation and maintenance of information systems. The ICASE tools and IPSEs, described in section 2.5, are examples of life cycle tools.

Integration across life cycle stages

"Vertical integration" or integration across life cycle stages reflects on the number of life cycle stages supported by the CASE tool. Individual tools become much more powerful when linked to other tools which support different stages of the development life cycle. If a tool only supports one or a few life cycle stages it is called a point tool. Point tools may well be useful in an isolated context, but they only offer support for part of the job, e.g., just the analysis stage or just the implementation stage. The recent increase in popularity of the life cycle products, relative to point tools, is a clear indication of a growing recognition of that a loose collection of point tools is less likely to offer effective and efficient support during the whole life cycle than what may be expected from an integrated toolset which has been designed to cover the whole development life cycle.

This dimension does not cover consistency checking issues. They are left to the next dimension; the degree of method support.
Degree of method support

This dimension discusses the level of method support implemented by the tool. We have stated above that tools support system development methods and not vice versa. This means that having decided on a tool one implicitly has decided on a method. Conversely, if one feels strongly about a specific method, one should make the choice of tool dependent on the choice of method.

There are three different dimensions that may be used to further characterise the integration issue; integration by user interface, integration by data, and integration by control.

User integration reflects on the way in which the functions offered by a tool are invoked. Tools should share the same "look and feel" so that a user can interact with an unfamiliar tool without difficulty.

Data integration reflects on the degree to which point tools agree on a common representation of data. Although it is not always possible to agree on a common representation of data for all tools, it may still be possible for clusters of tools to share common data. The degree of data integration determines the number of consistency checks that may be performed. Although consistency checking is weak in a large number of contemporary tools, the trend is towards more consistency checks. Still, many contemporary tools operate more as drawing editors than CASE tools. There are also certain restrictions on the degree of method support that is possible with contemporary system development models and methods: The models and methods supported by contemporary tools are to a large extent 10-15 years old and do not exhibit the constructivity property. The syntax and semantics of the various languages used for requirements capture, technical design, and implementation are different. This means that the development objects developed during the various stages do not relate properly to each other. A common representation of data is therefore difficult to establish. Verification and validation of specifications can then only be done by inspection, by prototyping, or by execution. E.g., it cannot be automatically checked whether the technical design of a development object satisfies the requirements to this object.

Control integration reflects on the manner in which tools are invoked. E.g., knowledge about the sequence in which tools must be applied to particular development objects may be used to guide the developer through a complex sequence of tool invocations.

Open or closed specification database

This dimension reflects on the openness of the tool, i.e. whether the input and output data formats of the tool is published and properly documented. An open specification database is a prerequisite to achieve data integration. The early point tools always maintained a proprietary data format, which made it difficult to integrate them with other tools. Even today the majority of tools operate on proprietary specification databases, although the specific formats may be public knowledge. The market pull is clearly towards open standards for the specification database.

Two different directions can be observed. The two directions are gradually merging into one: Firstly, we have the large players in the computer in-
dustry which try to promote their proprietary solutions as de facto "standards". This is done by forming alliances with other tool builders and publishing proprietary solutions within a limited community, and building an integrated tool environment which nobody can ignore. Secondly, the standardisation bodies (ANSI, ISO, ECMA, etc.) try to promote their standards. These standards are very much influenced by the large industrial players. Such standards seem to be unable to gain market acceptance before they are supported by the proprietary products from the large industrial players.

Support for systems development teams

Development of large complex information systems require the coordinated efforts of a team of systems developers. Large jobs simply cannot be done on time without the coordinated efforts of many persons. However, systems development in teams is often hampered by the lack of flexible communication and coordination among project members to allow true sharing of specifications.

In a team setting, different developers will be working on related sets of specifications. They will need to synchronise their work. Sometimes the specification sets will overlap, and the need for communication and coordination among the members increases. Introducing improved explicit control over specifications developed in a distributed environment will clearly improve on the productivity and quality of the work performed by the development team. This functionality requires that the tool is based on a specification database that provide shared access to specifications. Analyst/designer's workbenches and code generators rarely utilises a specification database, while life cycle tools are always based on a specification database.

Hardware and software platform

Hardware and software requirements deal with the overall configuration of networks, workstations, mainframes, and software platforms. They cover computers, peripherals, database management systems, graphics systems, operating systems, and communication software used, required, or provided by the tool.

To simplify the characterisation of this dimension we have chosen to classify the tools into workstation based tools (WS), mainframe based tools (MF), and tools which are client/server based (CS). The typical workstation based tool is either oriented to DOS/Windows, OS2/Presentation Manager, or Unix/Motif.

Market position

Sales information deal with market oriented pragmatic issues such as the prices for single copies and site licences, whether there is a discount policy available, history of sales figures, and the reference customer base.

The different categories of tools which we consider in this thesis covers a large spectrum with respect to the issues of this dimension. The spectrum goes from a low end represented by relatively cheap tools which may be
used on an experimental basis to gain experience and provide input to a more rigorous tool purchase analysis, to a high end represented by expensive tools which require full commitment from the whole development organisation to have effect and thus involves a fair amount of risk. The general tendency is that the analyst/designers workbenches and code generators comprise the low cost and low risk tools, while the life cycle products and repositories comprise the high cost and high risk tools.

We have chosen to classify into tools which have a growing (G), stable (S), or declining (D) market position. The desire for more integrated functionality drives a tendency in the market to move from the low cost/low risk tools to high cost/high risk tools.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Analyst/design</th>
<th>Code gen</th>
<th>Life cycle</th>
<th>Repositories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality</td>
<td>SIA</td>
<td>IA</td>
<td>SIAP</td>
<td>IAP</td>
</tr>
<tr>
<td>Life cycle stage focus</td>
<td>Front-end</td>
<td>Back-end</td>
<td>Many</td>
<td>Many</td>
</tr>
<tr>
<td>Integration across stages</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Method support</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Open/closed specification database</td>
<td>Closed</td>
<td>Closed</td>
<td>Closed</td>
<td>Open</td>
</tr>
<tr>
<td>Support for development teams</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Hardware and software platform</td>
<td>WS</td>
<td>MF</td>
<td>WS/CS</td>
<td>CS/MF</td>
</tr>
<tr>
<td>Market position</td>
<td>S</td>
<td>S/D</td>
<td>G</td>
<td>G</td>
</tr>
</tbody>
</table>

Fig. 2-7: Properties of categories

Summary of dimensions versus categories

This section gives a brief summary of the properties of the four CASE tool categories relative to the eight dimensions that have been discussed (figure 2-7). Average properties and average trends are given for each tool category and each dimension.

A principal weakness of this summary is the generalisation necessary to make it fit within the spatial limits of a table. Thus, it will always be possible to identify single tools within each category which by our definition is not typical and does not fit into our classification.

This chapter proceeds to describe each of the four tool categories identified in this section.

2.3. Analyst/Designer’s workbenches

The primary purpose of the analyst/designer’s workbenches is to assist in activities related to subject domain and implementation domain modelling. These activities involve identification of the problem, discussions with the user community to uncover the requirements to a problem solution, identification, naming, and representation of real world phenomena
of interest and formulation of a systems design that can act as input to programming activities and that is conformant with the requirements.

The output from the analyst/designer's workbench is a set of system specifications that describes the problem domain, identifies the problem, states the end user's requirements to a solution, and describes a solution that conforms to the requirements, and that can act as input to the subsequent programming activities.

![Diagram of analyst/designer's workbench](image)

**Fig. 2-8: Generic analyst/designer's workbench**

The number of tools that fits into this category is relatively large. It is however, a very homogeneous group of tools that exhibit similarities to a much larger extent than differences. We therefore choose not to describe any tool in particular, but accept the errors of generalisation. Well known examples include Excellerator by Index Technologies, StP by Interactive Development Environments, Analyst/Designer Toolkit by Yourdon Inc., and Blues by Interprogram BV.

The archetypical analyst/designer's workbenches consist of three different layers as depicted in figure 2-8.

**The user interface**

The user interface layer has previously to a large extent been based on home-grown systems developed by the tool vendors. The trend is clearly towards the three "survivors" in this market; Motif, Presentation Manager, and MS/Windows. The interface is based on a WIMP (Windows, Icons, Mouse, Pop-up menus) approach. The more specific interface for each analysis and design tool depend on the modelling concepts supported by the tools.
The analysis and design tools

The majority of analysis/design workbenches support several analysis and design models and techniques. Typical examples include activity/function decomposition diagrams, data flow diagrams, activity dependency diagrams, state transition diagrams, decision tables, structure charts, action diagrams, and entity relationship diagrams. The functionality for manipulating the diagrams is normally quite extensive including functions on objects like move object, remove object, add object, change object size, grid on/off/dot/line, annotate object, slide text, merge objects, split objects, stretch diagram, compress diagram, scale diagram, redraw diagram, refresh diagram, save diagram, archive diagram, and undo last change.

Thus the workbench consists of a selection of tools partially complementary and partially overlapping in concepts and scope. The majority of tools support the same selection of the techniques mentioned above dominated by the activity/function diagrams, data flow diagrams, structure charts, and entity relationship diagrams. However, the integration across stages and the method support is hampered by that fact that most of the models/techniques are 10-15 years old. And they usually do not offer formal verification or validation of the specifications. Some checks have been observed, e.g., basic drawing rules, checks to preserve meta model, and consistency checks like data flow diagram levelling.

The data dictionary

The concepts identified and captured by the analysis and design tools will all be stored in the data dictionary. The structure of the dictionary range from a document oriented approach to an integrated approach. In the document oriented approach the concepts are grouped into relatively coarse-grained chunks of data, e.g., a data flow diagram constitutes one document, with few relationships identified and recorded within each document as well as among the documents. In the integrated approach the granularity of the data being recorded is much finer, and the relationships between concepts are many and meaningful.

Concepts supported comprise entities (groups, partitions, distribution, covered, volumes), entity sub-types, attribute types, permitted values (codes, ranges), relationship types (degree, optionality, exclusivity), identifiers, activities/functions, data stores, events, messages, data flows, sources, sinks, external entities, and user responsibilities.

The dictionaries range from single user access systems to systems that allow concurrent access with password control, version control, locking, confirm before commit, undo after commit, recovery, and facilities for re-organisation.

Non-functional properties

The workbenches usually run on workstations. The trend is currently towards MS/Windows, Presentation Manager, and Motif. Early workbenches were typically standalone solutions, while the trend is definitely towards networked solutions which means more emphasis on the repository component.
As the ambitions of the analyst/designer's workbenches grow, the tools seem to put on weight. One particular trend is adding code generators, and thus moving from the analyst/designer's workbench category to the life cycle category.

2.4. Code generators

If you take a development project as far as the analyst/designer's workbenches will go, you still have no operational applications. The code generators would ideally take the output from the analyst/designer's workbench and generate a ready to run program. This does not happen automatically. Much manual intervention is necessary. Still, the tools that can be categorised as code generators, are products which generate programs with little or no conventional programming necessary.

Tools in this category fall into one of two types; code generators and fourth generation languages. The architecture for a generic code generator is depicted in figure 2-9.

Little or no domain knowledge is embodied in the code generators. The knowledge is based on the use of defaults "hard wired" into the tools rather than explicitly stating the knowledge, e.g., as rules. The tools are normally tailored towards a specific class of applications, e.g., transaction oriented systems based on relational databases. A narrowing of the class of applications supported has not yet become common. However, this may be handled by introducing domain specific application templates (e.g., banking, retailing) supplied by the tool vendor or by independent consultants.

The prime feature of the code generators and fourth generation languages is that they insulate the developer from the tedious and to a large extent repetitive details that the writing of operational code often involves. Details concerned with the handling of communications equipment, database systems, screen input/output, and report generation, often makes up the major part of traditional transaction oriented applications written in third generation languages. The heavy use of defaults can result in a "lowest common denominator" architecture for the applications. This may result in e.g., a "3270 interface" being used on applications running on PCs.

![Fig. 2-9: Generic code generator](image-url)
Chapter 2. The CASE tool domain

Code generators

Code generators produce source code in an established third generation language such as Cobol or PL/1. Once the code is generated, the code generator itself may be discarded, and the generated code executed and maintained as any other third generation language application.

Although one is generally advised not to modify the generated code, the freedom to do so if need be, reduces the tie up of the organisation to the specific tool and its vendor. This feature is of psychological rather than practical importance, since most developers will not go back to maintaining the third generation code, having once used the code generator.

Examples of code generators are APS, Delta, Goldrun, Interlagen, Netron/CAP, Sourcemanager, Telon, and Vax Cobol Generator.

Fourth generation languages

Fourth generation languages may be classified into either professionals' 4GLs or end-users' 4GLs.

The tools require the continued presence of a run-time component supplied by the tool vendor. The better tools now have compilers and will perform well on adequate configurations. However, applications developed with a tool in this group usually require more memory (and other computing resources) than do comparable applications developed in a traditional third generation language.

- Professionals' 4GLs have extensive facilities for professional developers of large applications. Examples of professional 4GLs are ADS/O, Corvision, CSP, Ideal, Mantis, Natural, Powerhouse, Pro-IV, and UFO.

- End-users' 4GLs are typically "information centre" products. They are typically used by managers and non-programmers to create simple decision support applications. Examples of end-user 4GLs are Focus, Mapper, Ramis, and SAS.

Specifying an application

In the code generation category of tools an application is specified through the application of one, two or all three of the following techniques:

Fill-in screen forms or menus guides the developer through a series of screen forms and menus as the tool collects the required information. Although it is generally easy to get started with tools employing this technique, applications tend to be constrained to the same software architecture. This is due to the fact that the products tend to build applications by inserting parameters in a pre-existing program skeleton.

Screen and report painting is another common technique, also found in some analyst/designer's workbenches. Developers may design screen and report layout without writing procedural code or specifying coordinates. The screen or report layout is rapidly done using a
mouse or keyboard to arrange fixed text and variable data fields on
the screen.

**Very high level languages** cater for the need exhibited by computations
that require the specification of complex branching logic and calcula-
tions. The very high level languages have an expressiveness al-
most identical to that of 3rd generation programming languages.
However, database and screen handling is simplified. The extent to
which the languages are to be used, depends on the nature of the ap-
lication and the orientation of the tool. Some tools have their empha-
sis on the two former techniques, while other tools are more like very
high level languages with powerful programming environments.

**The data dictionary products**
All the products in this category maintain a data dictionary of some kind.
The dictionary stores data definitions, rules and procedures, and screens
and reports designs. An active data dictionary control and impose consis-
tency on the database, applications, and other design components. Active
dictionaries control applications at run-time.

**Non-functional properties**
The traditional code generators run on mainframes, while most fourth
generation languages run on workstations.

An increasing demand for heterogeneous target environments can be no-
ticed. This results in a need for portability of the generated applications,
as well as the need to generate cooperative systems with components run-
ning on both workstations and mainframes. Cross generation facilities is
becoming a must. However, as most of the tools rely on a vendor supplied
DBMS, the portability of applications is limited to the portability of the
DBMS.

As is the case with analyst/designer's workbenches, code generators
seem to move towards life cycle products. Code generators do so by acquir-
ing analysis and design capabilities in the front.

**2.5. Life cycle tools**
The life cycle tools focus on giving support all the way from a diagram-
atically oriented systems analysis stage through to generation of code,
all through the development life cycle. The tools frequently provide project
management support and support for office functions. This category com-
prises the tools called Integrated CASE, ICASE, and Integrated Project
Support Environments, IPSEs. ICASE tools are single vendor integrated
tool packages with life cycle support. ICASE tools are often called "second
generation" CASE with front-end (or upper CASE tool) functionality and
back-end or (lower CASE tool) functionality integrated into one tool pack-
age. ICASE tools which are multi vendor are sometimes called C-CASE.
IPSEs are environments, frameworks, or "backplanes" which integrates
software development tools. IPSEs are also concerned with management
of the system development process.
Integration

The life cycle tools put strong emphasis on integration of the different models that are supported by the tool. The two most important dimensions of integration in this context are: horizontal integration among models used during the same life cycle stage, e.g., consistency checking between data flow diagrams and entity relationship diagrams, and vertical integration among models used during different life cycle stages, e.g., consistency checking between data flow diagrams and structure charts. The current trend is towards tight integration in both the horizontal and vertical dimension (figure 2-10).

Code generation

The code generated range from prototypes of various kinds, via skeletal structures that need to be completed manually, e.g., StP [76], to fully operational code, e.g., IEF [141]. The trend is clearly towards code generation
facilities that generates operational executable code. The job is not considered done until the result is an operational application.

The specification repository

The approach is strongly repository based with the ambition of maintaining a single canonical representation of the whole application where the different models used during the life cycle stages may be considered as logical views into this representation [51]. The repositories will store data on:

- the process of information system development and maintenance;
- the development objects created during that process.

This is done by implementing a meta model where these facets are integrated.

Closed and open ended tools

Some of the life cycle tools come as “ready wrapped” solutions where no provision for integrating third party point tools have been made. Typical examples are the so-called ICASE tools. Examples of ICASE tools are IEF by Texas Instruments, CorVision by Cortex, and Cadre-Teamwork by Cadre Technologies. The reader is referred to appendix A for a detailed description of Information Engineering Facility, IEF, by Texas Instruments.

Other life cycle tools are open ended and support integration of tools through the framework provided by the representation form in the repository. Typical examples are the so-called IPSEs. IPSEs and the closely related Software Factories are discussed in detail in appendix B.

2.6. Repositories

The repository tool category is a common denominator for concepts, architectures, research prototypes, and products that act as repositories for specifications produced by a variety of other (third party) tools.

The repository tools simplify the construction of information systems engineering environments by providing a set of commonly needed facilities, like integration components and support for higher level constructs that are not commonly found in operating systems. Another purpose is to support the porting of environments among different hardware configurations and operating systems.

Integration

The repository tools provide facilities for integration of tools and thus provide generic utilities that improves integration. Integration in this context has three prime facets: integration within life cycle stages, integration across life cycle stages, and integration across a distributed development environment.
Integration within life cycle stages focus on the various models used during one specific life cycle stage in order to enforce consistency within each stage. E.g., integrating data flow diagrams and entity relationship diagrams during the analysis stage will ensure consistency between flows and datastores in the data flow diagram and aggregates of entities and relationships in the entity relationship diagram. This aspect of integration will increase productivity and quality in the development life cycle.

Integration across life cycle stages focus on the specifications created by the different point tools that support different stages of the development life cycle. It is desirable that output from one tool can be automatically fed into another tool. This may be realised provided that the concepts manipulated by the tools can be related to each other. This may decrease the need for manual intervention and eliminate a possible source of errors, to increase productivity and quality in the development life cycle.

Integration across a distributed development project improves communication and coordination among development project members. Different developers are working on related sets of specifications. They need to synchronise their work. When the specification sets overlap, the need for communication and coordination increases. Introducing explicit control over specifications developed in a distributed environment will improve on the productivity and quality of the work performed by the development project team.

Special DBMS requirements for CASE

It can be helpful to think of a repository as a DBMS for CASE tools. In addition to all the features usually supported by a DBMS like non-redundancy of data, data independence, queries, real-time updating, locking, concurrency, integrity, etc., the CASE application require special attention to the following areas: (1) handling multiple versions of specifications, (2) maintaining dependencies among development objects created by different tools, (3) enforcing integrity constraints to ensure that the database remains consistent and meaningful, (4) supporting locks on data to avoid loss of work in case of update conflicts, and (5) maintaining database security through enforcement of access privileges.

Repository architectures

There are purpose built repositories and there are repositories implemented as applications on a standard commercial DBMS. The DBMS based approach has considerable attractions since the DBMS already provides much of the functionality required by the repository. However, most existing DBMS are not designed for CASE applications: the repository implementer will still have to do considerable work to extend the integrity, security, and recovery features of the DBMS. Version control, absent from most DBMS, will have to be added at the repository level. And the performance of, say a relational DBMS will probably not be optimised for the complex data manipulations required by CASE functionality. The current trend is illustrated in figure 2-11.
The users

There are four groups of people that need access to repository components; (1) the hardware platform suppliers need to provide the hardware and operating system platform to access to the resources of the repository, and provide the primitive functionality used by the repository, (2) the repository suppliers need to provide the platform components for interfacing between the user and the resources of the repository and provide most of the functionality of the platform, (3) the tool suppliers carry out the work of producing tools and generic utilities for the repository. This level of the repository provides the "value added" needed to make the repository into an effective problem solving environment, (4) the developers use the repository for solving various application problems. Designers, developers, managers, platform administrators, configuration controllers, secretaries use the customised repository to build target systems according to customers' information system requirements.

The reader is referred to appendix C for a detailed description of PCTE, and appendix D for a detailed description of IBM's AD/Cycle.

2.7. Trends in CASE tool development

The CASE tool market is very much driven from the vendors side. Among the current trends we would like to specifically mention four directions; a strive towards improved integration, the commercialisation of meta-CASE, reverse engineering, and reuse.

Improved integration

There are three interesting directions of integration; user interface integration, data integration, and integration of teams of developers.

In the area of user interface integration there has been a distinct shift from tools being based on an assorted collection of "homegrown" interface systems to Motif in the Unix environment and Presentation Manager and Windows 3.0 on PCs.
The enabling technology for data integration is first and foremost repositories. The repository field is dominated by a few central proposals: PCTE has recently been adopted as ECMA standard 149 [42] and is widely used within Europe. Hewlett Packard's SoftBench has been well accepted in USA, and HP is currently involved in collaboration efforts both with IBM and the PCTE community. IBM's AD/Cycle Information Model is already supported by several CASE vendors, and the number of vendors which has announced AD/Cycle compliance is quite impressive. Finally, DEC's CDD/Repository has attracted some interest, and some vendors have announced support for it.

Other aspects of data integration which are currently attracting attention are increased method support with rigorous and extensive design rule and consistency checking both within and across life cycle stages, full life cycle support all the way from analysis down to executable code, and cross life cycle capabilities which enable the developer to take full advantage of powerful engineering workstations.

The third aspect of integration concerns the integration of the members of a team of developers. LAN based development environments with a central repository is rapidly becoming the rule. The central repository may be physically centralised or distributed across the LAN. It is expected that multimedia technology will be integrated with the tool environments to provide effective tools for concurrent engineering over wide geographical distances.

MetaCASE

The commercialisation of metaCASE has produced tools that offer the ability to completely (re)define the models and methods used in the tool. This involves the definition of visual representations, design rules, and target code generation optimisation which altogether is a major undertaking.

The justification of metaCASE is that it is better to adapt a metaCASE tool to support the models and methods currently in use in the development organisation than to buy a good tool and adapt the working practise in the organisation accordingly. Hence, the costs associated with changing the working practise of the organisation to comply with the requirements of a commercially available tool are considered to be greater than the costs associated with building a CASE tool to support the models and methods currently in use in the organisation. Or, to put it differently, it is believed that the development organisation is better served by providing tool support for "any old model and method" that has been in regular use within a development organisation than to deploy a carefully selected set of models and methods supported by a mature tool in the same organisation. Bearing in mind the fact that the most advanced tools currently available in the market still have a long way to go before they can be called adequate, and that the development cost of these tools is in the multi million dollar range, it is our opinion that the justification for metaCASE does not hold for a system development organisation.

However, a metaCASE tool is of significant value to anyone who wish to be able to perform research on system development methods and models.
Reverse engineering

The large portfolio of existing "Cobol applications" makes reverse engineering an attractive market segment. Reverse engineering tools now support database derivations from physical to logical representations. The existence of interactive program understanding based on static and dynamic code analysis, together with facilities for extracting all code from an application related to a specific function so that it can be replaced or reengineered means that the reverse engineering field is rapidly moving away from its infancy.

Reuse

The software engineering community has been proponents of reuse for a long time. There seems to be a growing understanding that there are no simple solutions. Object orientation has not delivered as promised. The specific problems encountered during systems analysis require behaviour oriented abstractions of the development objects expressed in a domain oriented language coupled to extensive domain knowledge. For reusing this we currently lack both method and tool support. However, large-scale reuse in the form of application templates has emerged and seems to offer temporary relief.

In spite of the promise given by the trends described in this section, we would like to stress that CASE is still struggling to get out of its infancy. The following quote may serve as an illustration [151]:

"With the introduction of so much new technology, the biggest problems with CASE continue to be the fragmented nature of the tools and methods, and the need for more mature organisational frameworks in which to apply it."

2.8. Chapter summary

A major repositioning is currently taking place in the CASE vendor market. A large number of CASE tools are currently on the market. Few of these tools are providing acceptable integration across the development project life cycle, and across a distributed development environment. Hence the name point tools. An expected shake out in the market will leave only a few surviving point tools to continue the fight for market shares. The evaluation of the successful points tool will be dominated by technical issues more than financial strength of the manufacturer, provided the tool fits with an existing platform.

Among the platforms, the few tools that do provide acceptable integration across the development life cycle as well as across a distributed development environment, only a few can be expected to be successful in the market. Financial strength of the manufacturing company is more important than any other property, provided that the platform satisfies the basic integration criteria. This is so because a platform of this kind will be the single most critical part of any company's development strategy and information technology infrastructure. The platform as well as the systems developed using the platform, will record and maintain information of paramount strategic and operational value for the business. If the soft-
ware fails, the company will be out of business within days or weeks. One can therefore not afford to choose a platform, no matter how superior in technical quality, where there is a risk that the manufacturer is out of business in overseeable future.

Our research objective is to augment contemporary CASE with functionality for supporting system development performed by teams of individual developers. The various categories of CASE tools come out differently on this issue. Shared access to development data is dependent on whether the tool has a specification database or not.

Most analyst/designer workbenches and code generators do not have a specification database. Shared access to development data is therefore seldom supported. On the other hand, most life cycle products and repository products have a specification database and thus provide possibilities for members of a team to have shared access to the specifications. The shared access is implemented in much the same way as shared access to a conventional DBMS. Objects that are in the process of being modified by one developer are locked in the specification database, and may not be accessed by other developers.

Shared access to the specifications without the limitations exhibited by contemporary CASE should consequently be provided.
3. Cooperative work support

The study of organisation-wide systems that integrate information processing and communication activities, systems characterised as "the electronic workplace", is a new multidisciplinary field termed groupware\(^1\) or computer-supported cooperative work\(^2\) [59][82].

This chapter gives a state of the art survey of the field of cooperative work support. This comprises a discussion on the definition of the groupware concept as well as a characterisation of the group interaction process, followed by a presentation of two taxonomies of groupware. Finally, a survey of selected categories of groupware is presented.

3.1. Aspects of groupware

Most software systems only support the interaction between a user and the system. Even systems designed for multi user applications, such as office information systems, provide minimal support for user-to-user interaction. This type of support is clearly needed, since a significant portion of a person's activities occur in a team, rather than an individual, context. Three key areas are: communication, collaboration, and coordination [44].

- Computer-based communication is not yet fully integrated with other forms of communication. A separation exists between asynchronous, text-based communication such as electronic mail and bulletin boards, and synchronous communication such as telephone and face-to-face conversations. Applications such as voice mail and talk programs aim to bridge this gap.

- Effective collaboration demands that people share information. Current information systems insulate users from each other. E.g., designers working with a CAD database are usually not able to simultaneously modify different parts of the same object and be aware of each other's changes. What is needed are shared environments that unobtrusively offer up-to-date group context and explicit notification of each user's actions when appropriate.

- Coordination of a group's activities enhances the effectiveness of communication and collaboration. E.g., without coordination a team of programmers will often engage in conflicting or repetitive actions. Coordination can be viewed as an activity in itself, as a necessary overhead when several parties are engaged in performing a task.

We may say that groupware is software and hardware for shared interactive environments.

The term shared indicates that two or more participants interact with one another in such a manner that each person influences and is influenced by each other person. No upper limit in the number of participants is indicated, because mediated groups, as opposed to natural ones, can main-

\(^1\) The term groupware was coined in 1982 by Peter and Trudy Johnson-Lenz [84].

\(^2\) In a recent panel [60] Irene Greif commented that she (together with Paul Cashman) coined the phrase computer supported cooperative work as part of the work related to the call for an invitation-only workshop on CSCW held in Cambridge, Massachusetts in 1984.
tain joint awareness with very large numbers of persons. An objective of some groupware applications is to increase the awareness that a person is a member of a group, and increase the number of persons that can interact "as a group" [105].

The term *interactive* is used to indicate that time constraints are managed by the system. Many groupware applications appear to support real-time interaction [44][77]. Others merely enforce deadlines that can span weeks [116][50]. In either case, technical limitations on interaction is negligible in terms of the objectives of the application. Systems that exclude reference to real time are not considered to be groupware applications.

The term *environment* includes software and hardware that provides the context for interaction. Hardware may include specially designed furnishings and architectural spaces that are considered integral to the desired utilisation of a given software application. A groupware application may require a specific organisational environment. More powerful applications can adapt to, or overcome limitations of their social environments.

**Definition of groupware**

Before we venture into a definition of the term groupware itself, we present definitions of the two components of the term:

*Group* Two or more persons who are interacting with one another in such a manner that each person influences and is influenced by the other persons [130].

*Ware* (a) manufactured articles, products of art or craft (b) an article of merchandise (c) an intangible item (as a service) that is a marketable commodity [152].

![Common Task Dimension Diagram](image)

Fig. 3-1: Two dimensions of the groupware spectrum [44]

In accordance with the above the goal of groupware is to assist groups in communicating, in collaborating, and in coordinating their activities. Groupware may be defined as [44]:
"Computer based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment."

The terms common task and shared environment are the essential terms that characterise groupware relative to other systems. This excludes multi user systems where the users may not share a common task, e.g., time sharing systems. The "intensity" of both the common task and shared environment dimensions may be characterised as being low or high as depicted in figure 3-1.

Regarding the common task (goal) it is worth noting that while all groupware systems require some agreement among participants (at minimum that they should be jointly used), interactions can be predominately conflictual. Management of conflict is often a crucial feature of a groupware system. Voting systems are examples.

Note that Ellis' definition does not reflect on whether the common tasks are performed simultaneously or not. We may introduce the distinction between real-time and non-real-time groupware, meaning that groupware that specifically support simultaneous activity is real-time groupware.

Note also that the term groupware is restricted to computer based systems. The term was first defined by Johnson-Lenz to refer to a computer based system plus the social group processes involved [84]. Being primarily concerned with the system level issues, our definition follows the thinking of Johansen who restricts the term to apply only to computer based systems [82].

Group interaction process

One of the features of groupware is that a group of people is cooperating on a common task. McGrath's Circumplex model [109] is based on the fact that tasks differ in nature and that the cooperation patterns of groups differ accordingly (figure 3-2).

According to McGrath the work done by a group or an individual may be categorised into four different processes. The four processes are generate, choose, negotiate, and execute. As indicated in figure 3-2 each process comprises two tasks. The generate process comprises planning tasks and creativity tasks; the choose process comprises intellective tasks and decision making; the negotiate process comprises cognitive conflict tasks and mixed-motive tasks; the execute process comprises contests/battles/competitive tasks and performance/psycho-motor tasks.

Further, the eight tasks are either characterised by cooperation or conflict, and they are either conceptual or behavioural. E.g., cooperating group members strive to have their opinions accepted by the rest of the group. Conversely they need to make decisions if they are in a conflict situation. When the members of a group are in the process of generating solutions to a given problem, they generate ideas in the conceptual situation, and they would generate plans in the behavioural situation.

Each task done fit into one and only one category in the Circumplex model. At the same time, the tasks are related. Related tasks are "neighbours" in the model. E.g., the task generating ideas is related to
planning tasks and intellecitive tasks. This is also true for the processes. E.g., the process generate is related to the processes choose and execute, but is not directly related to negotiate.

Fig. 3-2: The Circumplex model [109]

3.2. Taxonomies of groupware

This section presents two orthogonal taxonomies of groupware: The time and space taxonomy, and the application level functionality taxonomy.

Time and space taxonomy

Groupware can be conceived to help a face to face group, or a group that is distributed over many locations. Furthermore, groupware can be conceived to enhance communication and collaboration within a real-time interaction, or an asynchronous, non-real-time interaction. This may be represented in a two by two matrix (figure 3-3).

Other dimensions, such as group size, can be added to this simple matrix [83].
Application level functionality taxonomy

The breadth of the groupware domain can also be illustrated by a taxonomy based on application level functionality. This taxonomy is not comprehensive and many of the categories overlap.

The taxonomy covers the categories: Message systems, multi user editors, group decision support systems and electronic meeting rooms, computer conferencing, intelligent agents, and coordination systems (figure 3-4).

3.3. Categories of groupware

This section proceeds to describe several categories of groupware: Message systems, multi user editors, hypertext, group decision support systems and electronic meeting rooms, computer conferencing, intelligent agents, and coordination systems.

Message systems

Computer based message systems support asynchronous exchange of textual messages between groups of users, e.g., electronic mail and computer conferencing. The proliferation of such systems has lead to the
"information overload" phenomenon [66]. "Intelligent components" are sometimes added to the message delivery system to ease the user's processing burden by automatic filing or re-routing of messages.

**Multi user editors**

Multi user editors, or joint editors, are used by group members to jointly compose and edit a document. The editors are either for asynchronous use, e.g., ForComment [116] and Quilt [50][96], or for synchronous use, e.g., Collaborative Editing System [61], and Shared Book [97]. The object being edited is usually divided into logical segments with concurrency and access control for each segment. Editors may provide explicit notification of other users' actions.

Editors for synchronous use apply WYSIWIS (What You See Is What I See) interfaces to document editing. They usually apply a telepointer which is a kind of global cursor. The telepointer enables one user to take control of the other user's cursors (or a special cursor used only for "telepointing") and point to a specific place in the document.

These editors also need to develop what is called a floor policy. The floor policy is a set of rules which makes it possible to determine who gets to write on the global screen next. The floor policy may be restrictive and allow only one user to act at a time, or it may be very liberal and put no restrictions on the number of simultaneous users. One end of this spectrum is represented by the tools which implement a rotating token. Only the user who has the token is allowed to act. This strictly limits the number of simultaneous actions to one. The other end of the spectrum is represented by the tools who rely on the informal social protocols that develops within a group to control what happens on the global screen.

We proceed to describe two examples; Quilt and GROVE.

Quilt is an asynchronous multi media document production system for groups of people [50][96]. The system supports multi media annotation of text, message transfer among the members of the group, computer conferencing facilities, and support for sharing of information among joint authors of a document. Quilt was designed to support the documentation part of research projects. A deliberate design decision was taken not to support synchronous editing or brainstorming sessions, including the use of a telepointer. The system supports sharing of multi media annotations (comments) to documents. Multi media annotations are either recorded speech or written text. Annotations may be private, public, or directed messages. Private annotations are readable only by the author of the annotation, public annotation are readable by all users involved in the document, and directed messages are sent from one user to one particular recipient. The users are allowed to modify and comment existing comments. Quilt will maintain the history of modifications and comments. Each participant plays a role relative to a specific document at a given time. Roles may be defined and modified. User access is granted to roles. A document in Quilt comprises a current base document together with a number of proposed revisions. The revisions may be swapped with (replaces) segments of the base document. Quilt provide asynchronous computer conferencing facilities with messages containing both written text and speech. Quilt aims to be independent of the text editor being used.
The GRoup Outline Viewing Editor GROVE is a real-time text editor designed for use by a group of people simultaneously editing an outline during a work session [43]. Within a GROVE session, each user has his own workstation and bitmap display. Thus each user can see and manipulate one or more views of the text being worked on in multiple overlapping windows on the private screen. A view is a subset of the items in an outline determined by read access. A viewer is a group window for seeing a contiguous subset of a view. Views and viewers may be private, shared, or public. A private view contains items which only a particular user can read, a shared view contains items readable by an enumerated set of users, and a public view contains items readable by all users. Group windows display views and indicates who is able to use the window and who is actually participating in the session at any given time. This information is provided by displaying images of the people who are members of the view along the bottom border of the window. As users enter or leave the session, their pictures appear and disappear in all appropriate group windows. Users can modify the underlying outline by performing standard editing operations in a group window. The effect of editing operations is immediately visible to all users in the session. Outline items which are grey rather than black on a particular user's screen cannot be modified by that user. Users can enter and leave a session at any time. When users enter a session, they receive an up-to-date document unless they choose to retrieve a previously stored version. A session terminates when there are no remaining participants.

Hypertext

Hypertext systems employ non-linear structuring of text, graphics, and other media. This makes the hypertext approach feasible for annotation and joint editing systems. Hypertext systems form linked network structures with data in the nodes and occasionally support typing information on the links. Hypertext documents resemble nets of connected nodes with each link between nodes denoting an association between the information held in the nodes.

Hypertext systems have been successful in structuring information in such a manner that a user's ability to process it is enhanced. The efficiency and effectiveness of users' information processing are improved by the appropriate hyperlinks. However, the interpretation of "appropriate" varies among individuals. Hyperlinks which fail to match the mental models of a specific user, i.e., hyperlinks that are not appropriate for this user, will also fail to improve this user's information processing abilities.

It is possible to view argumentation and cooperative editing systems as multi-user hypertext systems, where a hypertext document is constructed by a number of users adding nodes to the network in an independent manner. However, the provision of tools that deal explicitly with the interaction arising in collaborative hypertext settings is relatively recent.

Several hypertext systems put emphasis on supporting cooperation. Some important features are: (1) support for social interaction among cooperating agents; (2) support for cognitive aspects of cooperation; and (3) support for practicality in both types of interaction, e.g., [112]. However, hypertext systems have provided inadequate support for the resolution of conflicts
which arise when several authors are cooperating on the production of a document [37]:

- Hypertext systems provide limited organisation of the document. The system should provide a means of organising related sets of hypertext nodes and links.
- Hypertext systems lack partitioning of the document. The system should provide some mechanism which allows teams to work together in independent hypertext partitions without risk of interference. The independent partitions may be joined at carefully controlled intervals.
- Hypertext systems do not provide adequate version and configuration control. The users should have the possibility to build version trees of nodes and links. Additionally, it should be possible to build some form of configurations.

Contemporary hypertext systems are mostly used to support asynchronous cooperation and collaboration, but also systems with synchronous support for cooperation exists. Examples of hypertext systems include Notecards [148], Contexts [37], the Arizona Analyst Information System [105], CSILE [124], SEPIA [71], and World-Wide Web [113][17].

Group decision support systems and electronic meeting rooms

Group decision support systems (GDSSs) provide computer based facilities for the exploration of unstructured problems in a group setting [38][90]. There are tools for decision structuring, such as alternative ranking and voting, and for idea generation or issue analysis. Many of these GDSSs are implemented as electronic meeting rooms that contain several networked workstations, large computer controlled public displays, and audio/video equipment. The goal is to improve the productivity of decision making meetings, either by speeding up the decision making process or by improving the quality of the resulting decisions. A well known example is the PlexCenter Planning and Decision Support Laboratory at University of Arizona [10].

Computer conferencing

This category covers three areas: real-time computer conferencing, computer teleconferencing, and desktop conferencing.

- Real-time computer conferencing

Real-time computer conferencing allows a group of users, who are either gathered in an electronic meeting room or physically dispersed, to interact synchronously through their workstations or terminals. There are basically two approaches to implementing real-time computer conferencing software. The first approach imbeds a single user application in a conferencing environment that multiplexes the application’s output to each participant’s display [78]. The second approach is to design the application specifically to account for the presence of multiple users, e.g., Real Time Calendar [123] and Cognoter [133].

- Computer teleconferencing
Telecommunication support for group interaction is referred to as teleconferencing. The most familiar examples are conference calls and video conferencing. Newer systems provide workstation based interfaces to a conference [58].

- Desktop conferencing

The desktop conferencing systems combines the advantages of teleconferencing and real-time conferencing while mitigating their drawbacks. This method still uses the workstation as the conference interface, but it also runs applications shared by the participants. The modern desktop conferencing systems support multiple video windows per workstation. This allows for display of dynamic views of information, and dynamic video images of participants, e.g., the MMConf system [31].

We proceed to describe two examples of computer conferencing systems; MERMAID and TWS.

MERMAID is a multimedia workstation developed to support cooperation on development of documents [151]. It may be characterised as a multi user editor integrated with computer teleconferencing facilities for video and sound. MERMAID has 5 different windows: (1) A window to create and manipulate multimedia documents. (2) A global window. The text and graphics displayed in the global window is displayed on all workstations involved in the conference. (3) A private window. The contents of the private window is only visible on one workstation. (4) A video window which displays video images of one or several participants. (5) A status window which gives an overview of the participants, their geographic location, etc. The data processing and communication bandwidth requirements posed by large sized, high resolution, “true” colour audio/video conferences severely limits the proliferation of this technology. Consequently the affordable solutions presently offer insufficient quality. However, with the expected development in computer hardware and telecommunications, this situation is likely to be significantly improved within short time.

TeamWorkStation, TWS, is designed to bridge the gap between the personal computer, the desktop, and telecommunication technology [78][77]. The goal of TWS is to provide distributed users with a real-time open shared workspace which all team members can see, point to, and draw on simultaneously by using heterogeneous personal tools. TWS implements translucent overlay of individual workspace images by superimposing two or more translucent live video images of computer screens or physical desktop surfaces. The overlay function thus created allows users to combine individual workspaces, and to point to and draw on the overlaid images simultaneously. The TWS architecture makes use of a video camera to capture the face images of the users, and another video camera to capture the desktop images. Moreover it uses the two video images plus workstation screens to maintain three different windows, containing shared computer screen images, private desktop images, and face images respectively. The shared window may be used in three different modes; overlay, tele-desk/tele-screen, and computer-sharing mode. The overlay mode provides great flexibility, e.g., a video image of one user’s desktop may be overlaid another user’s screen making it possible to draw or point by hand on another user’s computer generated screen while being geo-
graphically dispersed. The tele-desk or tele-screen implements multicast-
ing of individual desktops or screens. The computer sharing mode allows
 colaborators to operate one computer by connecting their keyboards and
 mice to the computer whose screen is shared. Translucent overlay of video
 images provide users with rich semantics that they can easily interpret.
 Findings indicate that users can differentiate up to three overlaid video
 images without much difficulty. Two important benefits are: (1) Since
 TWS allows users to keep using their favourite individual tools while col-
laborating, there is no need to master new sophisticated groupware. (2)
The TWS multiscreen architecture allows users to move any application
 program window between the individual and shared screens just by drag-
ging the mouse. This makes it easy to bring data and tools in each per-
sonal computer to the shared workspace. There are also drawbacks with
 this approach, e.g., the results of the collaboration cannot be shared di-
rectly as it exists as a series of overlaid video images, and the quality of the
 overlaid video images is not as sharp as most computer displays.
 Experience indicate that the overlay mode, although suffering a few
drawbacks, provides a very flexible and effective support for collaborative
 work [77].

Intelligent agents
Not all participants in an electronic meeting are people. In general, intel-
ligent agents are responsible for a specific set of tasks, and the user inter-
face makes their actions resemble those of human users, e.g., Liza [57].

Coordination systems
Coordination systems approach the problem of integrating and adjusting
individual work efforts toward the accomplishment of a larger goal. The
systems allow the users to view their actions together with relevant ac-
tions of others, within the context of the overall goal. There are four types
of models in use: form, procedure, conversation, and communication-
structure oriented models.

- Form oriented models focus on the routing of documents (forms) in
  organisational procedures. While the majority of these systems ad-
dress coordination by explicitly modelling organisational activity as
  fixed processed [101], the more recent systems have made attempts to
  make process support more flexible [86].

- Procedure oriented models view organisational procedures as pro-
gammable processes. This approach was first applied to coordina-
tion problems in the software process domain and takes the view that
  software process descriptions should be thought of and implemented
  as software [117].

- Conversation oriented models are based on the observation that peo-
  ple coordinate their activities via their conversation. The underlying
  theoretical basis for many systems embracing the conversation
  model is the speech act theory [125].

- Communication structure oriented models describe organisational
  activities in terms of role relationships. E.g., in the ITT approach
  [68][69], a person’s electronic work environment is composed of a set
of centres, each centre represents a function. Within centres are roles that perform the work and objects that form the work materials for carrying out the function of that centre.

3.4. Chapter summary

This chapter has identified the groupware field as information technology for shared interactive environments which supports groups of people engaged in a common task or in the fulfilment of a common goal. Several categories of such systems have been presented briefly; message systems, multi user editors, hypertext systems, group decision support systems, computer conferencing systems, intelligent agents, and coordination systems.

We identify the multi user editing and the computer conferencing systems as being of particular interest to us. Multi user editing systems are used by group members to jointly compose and edit a multi media document thus giving true information sharing among the group members. The computer conferencing systems are using live video to enhance the quality of communication among group members. Some systems provide integration of video images from various sources thus providing mixed modes of cooperation where pointing and sketching by hand may be combined with computer supported operations on a document.
4. Version and configuration management

Information system development results in a large number of development objects, e.g., administrative documents, conceptual models, test data, and computer programs. The various development objects will typically be modified several times during their lifetime. Information system engineering is characterised by sequences of modifications of the system specifications. Most systems will also undergo modifications after installation.

A software product influences and is influenced by many factors when it is used, e.g., other applications, users, public law, and operational rules. These are all subject to change and often forces modifications upon the software product [22]. Information systems are usually of critical importance to the business of a company. It is therefore of vital importance that the systems are continually adapted to reflect, e.g., changing business strategies.

This chapter is structured as follows: Firstly, we define version and configuration management. This includes the identification of important dimensions in version and configuration management. Each dimension is discussed briefly using examples from tools or models proposed in the literature. Secondly, a selection of approaches to version and configuration management is surveyed. The selected approaches are file based systems, SCM for engineering databases, and SCM in commercially available CASE tools.

4.1. Version and configuration management features

This section gives a characterisation of systems for version and configuration management from two different perspectives: On one hand we discuss definitions of version and configuration management. On the other hand we discuss different dimensions which may be used to characterise tools and approaches of version and configuration management.

4.1.1. Definition of version and configuration management

Configuration management is the discipline of controlling the evolution of composite systems. The purpose of configuration management, CM, is to control and maintain an evolving system, with both planned and unanticipated variability. CM is a well established discipline in many traditional engineering disciplines, e.g., mechanical and electrical engineering. CM is basically seen as a project management technique, and often takes the form of standardised procedures described in voluminous manuals. Important objectives are to constrain the development process, to make it manageable, and to extract project and product information for managerial control.
Software configuration management

Software configuration management, SCM, is the application of the goals and ideas of classical configuration management to software production. A software product is its own description and can be directly subjected to configuration management, while for physical items, only the descriptions of the system and its components are subjected to configuration management [98].

SCM may be viewed as version control for complex objects. Versioning requires the capability to store multiple versions of the same development object. A version of an object is considered to be an object in its own right. The relationship between two versions is either a revision or a variant. A revision reflects a development history where the most recent version replaces its predecessor. A variant reflects a development history where two versions co-exist without any one replacing the other.

The technical role of SCM is emphasised by Babich [11]:

"CM is the art of identifying and controlling modifications to the software being built by a programming team".

Another definition taken from an IEEE standard puts emphasis on the project management aspects [74]:

"A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to these characteristics, record and report change processing and implementation status, and verify compliance with specified requirements".

4.1.2. Categories of version and configuration management approaches

A number of version and configuration management approaches have appeared over the last 10-15 years, including separate tools which support parts of the problem, e.g., RCS [146], SCCS [119], Make [48][49], and integrated tool environments which aims at supporting a broader range of the problem, e.g., DSEE [94] and Adele [15]. The development objects that are most often controlled by these systems are text files, usually computer programs. More recently, systems for the versioning of structured objects have appeared, e.g., Damokles [39][40] and Orion [87].

The various tools and methods for SCM may be split into three categories: File based systems, SCM for engineering databases, and SCM in commercially available CASE tools.

The file based systems perform version and software configuration management on files, mostly text files. They include systems which are based on versioning of simple text files, as well as versioning of structured systems where a derivation graph models the structure of the system product.

A number of models for SCM for engineering databases have emerged. The engineering databases provide tool and database support for engineering tasks, e.g., CAD and software engineering.
Methods of SCM have to some extent been introduced in commercially available CASE tools. Towards the end of the chapter we take a brief look at support for version and configuration control in selected contemporary CASE tools.

4.1.3. Dimensions of version and configuration management

This section identifies the issues that should be addressed by a version and configuration management system. The survey is compiled from a number of sources of which [33][18][135] have been most influential. The issues surveyed are: Development object identification, development object versioning, efficient storage of versions, system regeneration, development object status accounting and auditing, team work, change control, process management.

Development object identification

Development objects must have a unique identification that can be qualified by version names. Classification, storing and access of the development objects should also be provided.

Development objects handled by contemporary CM systems are mostly text files, although some systems support the versioning of complex objects. Since most derived elements are easy to produce, usually only source elements are versioned [154].

Development object versioning

In an ideal world, only one version of each development object would be needed. In practice many concurrent versions are needed. The versions should consequently be easily retrievable, and use minimal extra storage.

Techniques for keeping track of versions of individual development objects are offered by both RCS [145] and SCCS [119]. Versions are either revisions representing development histories, or variants which represent co-existing versions. Version numbering follows a two level scheme, for major and minor versions. While the major version identifier indicates functional changes, the minor version identifier usually indicates less fundamental changes, e.g., correction of errors. Consider for example the development of Microsoft Word for Macintosh, where the shift from version 4.0 to version 5.0 involved major changes in functionality. The subsequent introduction of version 5.1 did not involve much new functionality, but the majority of the errors in version 5.0 had been fixed.

Chains of revisions consist of versions that are meant to replace one another, are usually termed version families, or variants. This indicates that two different variants are alternative version families [119][94][95][70][102][144]. A particular version (of a variant) is distinguished from other versions, and may be selected among other versions, based on a number of attributes. Attributes are user defined descriptors indicating characteristic properties of a version. The relevance of these attributes relative to the actual changes done from one version to its successor is highly subjective, since the purpose of the attributes is to "trigger the mind of the developer" [145][146].
Efficient storage

The problem of efficient storage of versions of software objects relates to the problem of saving disc space. The assumption is that the real difference between one version and its successor is relatively small compared to the total size of either of the versions. It is therefore assumed to be more efficient with respect to storage, to save a version and the difference between the version and its successor, than to save both of the versions in entirety. E.g., version 1.2 = version 1.1 + δ1, version 1.3 = version 1.2 + δ2. [119].

Application of this strategy in DSEE, [94], reports disc saving ratios ranging from 1:25 to 1:50. I.e. 50-100 versions may be stored in the same space that is required for two versions in full.

When the number of versions grows large, the time needed to access an arbitrary version becomes unacceptable if it must be regenerated through a sequence of applications of δs starting from the first fully stored version. Various schemes may be devised, e.g., assuming that the last revision is more frequently accessed than previous versions one may work from the end of the revision chain applying reverse δs. One may also apply reverse δs and forward δs in combination as the revision tree is traversed, and one may imagine storing every n-th version in its entirety to shorten the time needed to access an arbitrary revision, etc. [145][146].

Deltas in SCCS are embedded within the development object. RCS uses forward and backward deltas in combination. The last version on the main version family, the “trunk” of the revision tree, is stored in full, and previous revisions can be constructed with backward deltas. Forward deltas is used for branches.

System regeneration

System regeneration comprises identification and maintenance of relationships between development objects and selection of valid and consistent versions of the software for regeneration of a system. Mechanisms for optimising efforts for building the system should be provided.

Make [48] is a program for effectively rebuilding configurations, specifying the dependencies between development objects, and recompiling by comparing time stamps. Other systems use smart compilations [147] by using more fine grained dependency information.

In DSEE a version map is used to build valid configurations from different versions of development objects. A system model provides a record of the development objects going into a configuration. A configuration thread describes how to select versions for the development objects. In Adele [15] configuration selection is performed based on attributes and selection predicates. A configuration is specified by a selection predicate and constructed by traversing the configuration structure, adding development objects satisfying the selection predicate.

The configuration management assistant, CMA [118], provides configuration construction and validations. The system can detect if a certain configuration is applicable, i.e. complete, unambiguous, and consistent. An
algorithm for finding correct configurations has been devised by Agrawal and Jagadish [1].

Development object status accounting and auditing

System revisions are developed after proposals for change have been analysed. In order to monitor the progress in developing a system and its components, one may introduce state values for characterising the development state. Hence, development object status accounting and auditing comprises recording and reporting the status of development objects and gathering statistics about development objects. Consider the following set of state values which has been derived for a typical waterfall life cycle oriented method [8]:

- State values for determining a document's development state are:
  Initiated, Ready for Approval, Conditionally Approved, Approved, Delivered, Retired, Scrapped.

- State values for determining a subprogram's development state are:
  Initiated, Designed, Implemented, Unit Tested, Integration Tested, Approved, Delivered, Retired, Scrapped.

- State values for determining a system's development state are:
  Initiated, Specified, Designed, Programmed, Module Tested, System Tested, Approved, Delivered, Retired, Scrapped.

Traceability may be provided by maintaining a log of the work performed to come from one revision to the next.

Team work

When many users work together in a team, there must be some way of controlling the work and the interactions between them. This includes access control, locking, separated workspaces, merging of changes, and notification and co-authoring mechanisms.

Both SCCS and RCS use check-in/check-out mechanisms to avoid simultaneous update of the same development object. However, different variants may be worked on by different persons at the same time. Workspaces are used in several tools to prevent users to interfere with each other. Some approaches differentiate between two types of workspaces, the user workspace where local changes are taking place, and the project workspace, consisting of the project database and relevant libraries. Mechanisms exist to enable the transfer of information between the two workspaces and access according to user role and status of the development object.

The transaction notion of NSE [47] represents a coordinated unit of work. It reflects the structure of development objects and supports isolation of work, interactions between users, and merging of modifications. Users work independently in their environments, modifying the same or different configurations. When a user updates the repository with a new version, NSE checks to see whether the changes conflict with what currently exists in the repository. If a conflict occurs, NSE notifies the user and provides assistance in eliminating the conflict.
The Sherpa Design Management System provides a repository for files that are distributed on different hardware platforms [36]. The distribution is transparent to the users, and the system enables a team of geographically dispersed users to work on the same configuration of files. The Distributed Version Control System, DVCS, [89] supports distributed building of the system, using load balancing heuristics to reduce building time.

**Change control**

Change control and change requests are techniques for controlling the release of a product and the modifications to it, throughout the development cycle. This is done by having controls in place that ensure consistent software via the creation of a baseline product. Before modifications are performed, an impact analysis has to take place, and all developers influenced by the modification have to be notified.

In EPOS [28][27], versioning and configuration management are kept orthogonal to the module structure, by the implementation of change-oriented versioning, COV [67][98][111]. A complete functional change which may influence several development objects is described by a single global option. Configuring a system consists of selecting a set of mutually compatible changes.

Aide-De-Camp support the notion of a change set [132]. A change set comprises: the reason for the change, all related file changes, and details of who made the change and when. A change set represents a *logical* change to the product. To create a configuration the user selects a baseline and applies selected change sets to it.

LIFESPAN represents a change request by applying a series of forms, and the process of change is represented as a series of states, tasks, and roles [154]. A design change details out which development objects that must be modified and how the modification should be done. LIFESPAN uses this information to find the people that are affected by the modification. They are notified and have to agree on how the modifications are to take place. When the modifications are implemented in the new version, the person who requested the change in the first place, is notified.

**Process management**

Process management provides life cycle support and support for organizational policies. It must be possible to identify tasks to be done, how to perform the tasks, and estimated or required time of completion. It must be possible to adapt the model to each specific project throughout the course of the project.

Several approaches exists for process modelling: Rule based approaches may either be based on production rules, e.g., Marvel [85], or triggers, e.g., Adele2 [16], Task net approaches, e.g., Petri Net based [12], or Process Programming Languages, e.g., VPL [131]. Some systems adopt a hybrid strategy, e.g., EPOS [29]. In ISTAR the processes are modelled as a hierarchy of *contracts* [41]. A *contract* contains information about the information flow, the start of tasks, the completion of tasks, and the passing of results from the tasks.
This chapter proceeds to survey selected approaches to version and configuration management. The survey has been structured according to the previously identified categories file based systems, SCM for engineering databases, and SCM in commercially available CASE tools.

4.2. File based systems

File based systems provide software configuration management for files. The category includes standalone versioning systems which are restricted to version handling of text files, e.g., SCCS and RCS. The category also includes systems based on derivation graphs which model the structure of the components of software systems, e.g., Make, DSEE, Huff's model, Lockeman's model, and Tichy's AND/OR graphs.

4.2.1. SCCS - The Source Code Control System

The Source Code Control System was developed at Bell Laboratories in the years from 1972 to 1974 [119]. It was later included in the Programmers Workbench.

SCCS maintains arbitrary text files and provides a number of features for keeping version control of such files: incremental change, economy of disk space, check-in and check-out of revisions, identification keywords, and change documentation.

Incremental change model

The incremental change model assumes that a text file undergoes successive revisions in strict time order.

The incremental change model identifies revisions by release numbers and level numbers for each release, i.e. a two level structure, figure 4-1. A release is normally taken to identify a major change, such as addition of new features, where as level changes are normally associated with minor changes, such as correction of errors.

![SCCS version history](image)

Fig. 4-1: SCCS version history

In early versions of SCCS new revisions could only be inserted at the end of each release. This was later developed into a general tree structure, where any revision may be checked out for editing. Consider the linear version history depicted in figure 4-1. If version 1.3 was checked out for editing, the new revision would be named 1.3.1.1 as depicted in figure 4-2.
A configuration management approach for supporting cooperative information system development

Fig. 4-2: Version history with two branches

This new revision could be revised several times as depicted in figure 4-3.

Fig. 4-3: Two branches in parallel

Suppose now that revision 1.3.1.2 had to be revised, it would be checked out for editing and a new branch would be started. The first revision of this branch would be called 1.3.2.1 as depicted in figure 4-4.

Fig. 4-4: Version history with three branches

Thus the identification (numbering) of the revisions contains two or four components: Release, level, branch, and sequence number. However, the numbering scheme does not contain information to deduct revision history. It is not possible to infer that 1.3.2.1 is a revision of 1.3.1.2. The only semantics contained in the branch number is that it is the chronologically second branch created in the subtree whose root is the revision 1.3.

Economy of disc space

Economy of disc space is provided by storing all revisions of a file in a special library file, called the revision file. Text common to more than one revision is not duplicated. All revisions of the file are retrievable from the library file.
Check-in and check-out

SCCS provides check-out and check-in of revisions. A developer creates a new revision by first checking out an existing revision for editing. The check-out command locks the checked-out revision and produces a working file. The working file may then be modified. The versions that have been checked out for editing are locked and may not be checked out by anybody else. When the new revision is completed the check-in command stores the new revision in the revision file, unlocks the checked out revision, and deletes the working file. Check-in may take place any time, be it 5 minutes or 5 days, after check-out.

Identification keywords and change documentation

SCCS provides a number of keywords that may be inserted in the text. The keywords are expanded and replaced with current values when the text is retrieved from the revision file. There are keywords to identify e.g., modules, file names, revisions, check-in date and time, and line numbers. The keyword feature may be used to insert identification information in executable programs to facilitate error messages of the kind: "Internal error in module ... line ...".

SCCS also records information about the changes from one revision to the next, e.g., date and time of change, by whom the change was made, what has been changed, and why the change was performed. The motivation for the change is the only one of these that cannot be automatically recorded.

4.2.2. RCS - The Revision Control System

The Revision Control System, RCS, is essentially a redesign of SCCS, and developed by Walter F. Tichy [145][146]. RCS provides much the same facilities as SCCS, like incremental change model, protection facilities, storage economy facilities, and change documentation. It is claimed to be easier to use, faster in use, provides retrieval by attributes, and supports symbolic revision names [135].

The most important reason why RCS is faster in use than SCCS is that while SCCS use forward δs that are merged, RCS have chosen to use reverse δs which are kept separate. The main motivation behind this is the fact that recent revisions are more frequently accessed. Both tools use the δ generated by the Diff algorithm [110]. The Diff algorithm produces a δ which contains the differences between two text files. With reverse δs the last revision is saved in full, and δs are applied backwards to generate previous revisions.

Retrieval by attributes allows the developer to retrieve revisions not only by revision number, but also by, e.g., check-in time, author, and state.

While the revision numbering of SCCS has four number components, RCS adds two components for each added branch. The SCCS revision tree of figure 4-4 is shown in RCS style in figure 4-5.
RCS also provides symbolic revision names by a mapping of symbolic names to sequences of integers in the revision identifier. Revisions have an even number of integers while branches have an odd number of integers. All symbolic names in an RCS file must be unique. A branch or revision may have several different names associated.

4.2.3. Make - A Program for Maintaining Computer Programs

Make was developed at Bell Laboratories in 1975 [48][49]. Make regenerates systems when any of the objects that are prerequisite for the generation of the system have been modified.

The objects handled by Make are files and members of Unix libraries. Both of these object types have similar attributes such as owner, protection mask, last modification time. MAKE is based on the notion of a target file being created from one or several prerequisite files.

A target file may be an executable file having object code files as its prerequisite files. The object code files may have some source code files as their prerequisite files. A graph is formed that indicates the dependencies between the component files that are needed to regenerate a certain file. An example of such a graph is depicted in figure 4-6.
In an ordinary situation a system generation procedure for the example given in figure 4-6 may read

```
FORTRAN Read.FOR > Read.O
FORTRAN Compute.FOR > Compute.O
FORTRAN Write.FOR > Write.O
LINK Read.O Compute.O Write.O > Program.EXE
```

With Make the system generation instructions, the *Makefile*, would read

```
Program.EXE: Read.O Compute.O Write.O
    LINK Read.O Compute.O Write.O > Program.EXE
Read.O: Read.FOR
    FORTRAN Read.FOR > Read.O
Compute.O: Compute.FOR
    FORTRAN Compute.FOR > Compute.O
Write.O: Write.FOR
    FORTRAN Write.FOR > Write.O
```

As can be seen from this example, the derivation dependency graph is documented in linear text form in a Makefile. The Makefile lists all the targets together with their prerequisite files and the operations that have to be performed on the prerequisite files to regenerate the targets.

Make uses time stamps as provided by the "last modification time" attribute offered by most operating systems, to determine whether regeneration of a target is necessary. The algorithm to determine whether a target file is up to date, i.e. whether regeneration is necessary, is

A target is up to date iff
1. The target itself exists
2. All its prerequisite files are up to date
3. None of its prerequisite files are strictly younger (according to the time stamp) than the target itself

If a target is not up to date it is regenerated, thus made up to date, by executing the associated commands given in the Makefile. In the example the command

```
make Program.EXE
```

would result in the regeneration of *Compute.O* and *Program.EXE*. The traversal of the Makefile would trigger the command

```
FORTRAN Compute.FOR > Compute.O
```

since *Compute.FOR* is strictly younger than *Compute.O*. This would in turn make *Compute.O* strictly younger than *Program.EXE* which in turn would trigger the command

```
LINK Read.O Compute.O Write.O > Program.EXE
```

which would complete the regeneration procedure and provide an up to date version of *Program.EXE*. 
4.2.4. DSEE - The Domain Software Engineering Environment

The DOMAIN Software Engineering Environment [94][95] is a distributed software development environment that runs on Apollo workstations. There is one instance of DSEE running on each workstation.

One of the design goals of DSEE was to be able to work with any language, text processor, or editor. This was accomplished by implementing parts of the system directly in the Apollo AEGIS operating system. Versioning is integrated in the file system such that normal file system operations (open, read, etc.) apply to versioned files as well as files that are not versioned. The open command automatically consults the version map associated with a versioned file. Thus, tools (compilers, editors, text processors, etc.) do not have to be changed in order to "understand" versioned file formats. However, it also makes DSEE difficult to port to other platforms than the proprietary Apollo AEGIS system.

DSEE consists of five managers:

The History Manager provides text revision control much like SCCS and RCS.

The Configuration Manager provides regeneration facilities of systems much like Make.

The Monitor Manager provides and monitors user defined semantic dependencies between objects. It alerts the user when changes have been made to objects that one of his own objects depend on.

The Task Manager provides a way of defining tasks and subtasks in a software project. It cooperates with the History Manager so that physical changes made to text files are attributed to a logical change of the entire system.

The Advice Manager provides advice on how to accomplish tasks that are similar to tasks already performed. It uses experience data gathered from previous projects.

We proceed to look briefly into the History Manager and the Configuration Manager.

History manager

The history manager provides roughly the same functionality as RCS and SCCS. While both RCS and SCCS uses the Diff algorithm [110] to generate δs, DSEE uses Heckel's algorithm [63] to generate the δs. Statistics indicate that for a typical 3rd generation language module, each revision adds 1-2% to the size of the revision file.

DSEE uses merged δs which makes it possible to retrieve a revision by a single pass of the revision file. All programs can therefore read transparently any revision of a file directly from the revision file.

Configuration manager

The configuration manager uses a system model file which describes systems by giving the names of all components and dependencies among the components much the same way as in Make.
Like in Make, the system model file contains the commands for each derivation and is therefore able to support arbitrary derivations and not just the standard compilation or linking type of derivations. Components are named without version designations, thus yielding the need to bind versions at a later stage. Unlike in Make, the names of derived components are not given. It is said that "binaries are referred to as the result of translating the corresponding sources" [95]. It is hard to see how this strategy can suffice in a software engineering environment where the diversity of derivations is large and where there is a need to refer to derived objects.

In order to bind names of components in the system model file (the system description) to specific revisions DSEE uses so-called **configuration threads**. A configuration thread is given for each user. A configuration thread is a table giving the revision wanted by a user for each named components in the system model file. The choice of revisions may be stated explicitly, by branch, or by default.

A bound configuration thread, BCT, is a record of each derivation together with the commands necessary to perform the derivation.

### 4.2.5. Huff's Model

Karen E. Huff proposed an Entity Relationship data model for controlling systems of software files in 1981 [70].

The model is based on derivation graphs within a component hierarchy. Components are either **Initial Components** or **Derived Components**. All components are uniquely identified by a **Configuration Item Number**.

Initial components are either **Include Modules** or **Source Modules**. The model supports both parallel and sequential versions of initial components. Initial components are uniquely identified by a combination of the attributes **Name**, **Parallel Version**, and **Historical Revision**.

Derived components are either **Object Modules** or **Load Modules**. The object and load modules are derived components which are built by tools. The subschema for configuration identification is depicted in figure 4-7.

Huff argues that an exact record of each derivation in the derivation graph must be kept. The record must include the derivation tool used, its revision, and the exact options used for the derivation. Such a record is called a **derivation summary**.

The **Name** attribute of **Initial Component** is used to give a convenient (module) name and is not necessarily unique. However, the **File Name** attribute of **Component** is unique over the file system and used by all tools that access the component.

The model also includes a subschema for controlled evolution of a system (figure 4-8).

The concept of **Workspace** is central to this scheme. The workspace comprises tools and the view of the specification database which is accessible to the **Programmer** who is assigned to the workspace. Only one programmer is assigned to a workspace at a time. A programmer may on the other hand have several workspaces.
To be able to modify a revision, the programmer will have to lock the revision chain for an indefinite period of time. This is done by a Key held by the workspace. Thus, the key owned by the workspace locks a certain revision chain as long as it is being updated in a workspace.

Workspaces also have Contexts associated. A context is a view of the database which allows certain components to be seen by the workspace while some are invisible. Normally, a context is initially assigned to a workspace, and the programmer using the workspace adds new components to the context as work progresses.

Finally, tools are associated to workspaces. If a tool (compiler or linker) is assigned to a workspace, the programmer assigned to the workspace can use this tool. Note that Huff's model only treats compilation and linking of software modules. It is claimed that the model may be extended to cover any language and documentation tool.
4.2.6. Lockeman’s Model

Peter C. Lockeman presented his Entity Relationship model of version and configuration control in a software engineering environment in 1983 [102]. The model is based on viewing the software engineering model as a production line where a software product passes through a number of “workstations” on its way to completion.

Examples of workstations are requirements analysis, functional specification, preliminary design, detailed design, structured programming, code editing, verification, compilation, local testing, linking, and integration testing. Each workstation is represented by an executable tool which produces some output, a subproduct, which must be kept for possible later input to the next workstation.

M-graphs

The order in which tools are combined for consuming and producing products and subproducts, is determined by a derivation graph, the so-called M-graph. The M-graphs represent the production line through sequences of connections of tools. Tools are connected using control flow constructs for sequencing, iteration, decomposition, selective or collateral branching.

Consider the following example: Development of a text editor starts with functional specification and modular design before the design is decomposed and the individual modules are edited in a code editor before they are compiled. Compiled code goes through both path analysis and functional testing before all modules are linked to form an executable text editor. The M-graph of this tool set-up is shown in figure 4-9.
Product structure

A tool sequence determined by an M-graph induces a structure on the products that are developed using it. The corresponding structure is depicted in figure 4-10.

The figure shows the product structure of a text editor including functional specification, design document, and source code, object code, and tested object code subproducts for the three modules that comprise the text editor.

The subproduct graph has a single starting node and a single terminal node.

Versioning schema

The M-graphs and the subproduct graphs do not include versioning. To cater for the fact that each subproduct may exist in a number of alternative versions and that each version in turn may exist in several revisions, an Entity Relationship model has been developed, figure 4-11.

On the highest abstraction level, a system is viewed as a family of software products. Each family member is a product which consists of a number of subproducts. E.g., the text editor in the above example consists of subproducts such as installation manuals, user manuals, source code, object
code, etc. All parts of a system are considered to be subproducts, and each can be associated with a tool.

The next level of abstraction describes configuration of software products for particular environments, users, markets, etc. Configurations are built by selecting specific versions of the subproducts comprising the software product. Not all subproducts of a software product need to be included in a configuration.

![Diagram of subproduct structure]

Fig. 4-10: Example of subproduct structure [102]

The lowest level of abstraction is called the object. The object is a deliverable system configuration which is composed by selecting an instance (a revision) for each version.

The relationships unit, constituent, and component all deal with composition of products from subproducts, product configurations from subproduct versions, and product configuration objects from subproduct version instances respectively.

The relationships derivation, consequence, and continuation define the subproduct graph and the corresponding graphs on the version and instance levels respectively. For a given product the version and instance graphs are isomorphic with the subproduct derivation graph. The graphs define exactly which subproduct versions and instances that have successfully been used in actual derivations.

The isomorphism among the derivation graphs on the subproduct, the version, and the instance levels respectively seems to be a severe restriction on the ability of this model capture real life situations. E.g., to be able to provide alternative versions of the aforementioned text editor for say VT100 terminals and bit mapped display units, it may be necessary not
just to provide alternative implementations of some subproducts, but also
to provide different subproduct structures.

![ER Schema Diagram](image-url)

Fig. 4-11: The ER schema [102]

It is also highly unlikely that a derivation graph would not change over
time as development and maintenance continues. E.g., the text editor
might evolve from the three module architecture depicted in figure 4-10 to
an architecture with four modules.

4.2.7. Tichy's AND/OR Model

The AND/OR model for programming support environments was pro-
posed by Walter F. Tichy [144]. The model is based on an AND/OR graph.
The AND/OR graph is a directed acyclic graph, DAG, in which each node
is either a leaf node, an AND node, or an OR node.

Leaf nodes represent the primitive objects, e.g., individual text files. OR
nodes represent a version group. From an OR node at least one of the suc-
cessors must be chosen. AND nodes represent the combinations of all of its successors. This combination is called a configuration.

As an example consider a system S which consists of the two components A and B. Revision 1 of the system, denoted S(1) for short. It consists of revision 1 of component A and revision 1 of component B (figure 4-12).

![Figure 4-12: System configuration, revision 1](image)

The corresponding AND/OR graph is shown in figure 4-13. AND nodes are labeled "&" and OR nodes are labeled "+". The figure is interpreted in the following way: The root is an AND node representing the configuration S. Both successors A and B of S are therefore chosen. From the OR node A at least one of its successors are chosen (there is only one!) which means that revision 1 of component A is chosen. In the same way one must chose at least one of the successors of the OR node B, which means that revision 1 of component B is chosen.

So, from the AND/OR graph of figure 4-13 one can derive only one system configuration.

Suppose now that system S is modified. Revision 2 of S introduces revision 2 of component A and leaves component B unchanged. S is further modified, and revision 3 of S comprises revision 2 of A and revision 2 of B. The two configurations, revision 2 of S and revision 3 of S respectively, are shown in figure 4-14.

From the corresponding AND/OR graph (figure 4-15) it is possible to derive $2^2 = 4$ different configurations of S although only three have actually been proposed.
Consider now that our example system $S$ is further modified through the introduction of a new component $C$ (figure 4-16). The corresponding AND/OR graph for system $S$ after the introduction of revision 4 of $S$ represents $2 \times 3 + 2 \times 3 \times 1 = 12$ different configurations, although only four have actually existed (figure 4-17).

This illustrates a major problem with the AND/OR graph approach. As the number of component revisions increase, and the number of components increase, the number of configurations that are possible to derive from the AND/OR graph increases very much. Only very few of these configurations will be meaningful. The AND/OR graph contains no information to identify them.

To remedy this problem there are three mechanisms:

**Cutoff dates** Nodes are time stamped. A cutoff date for a configuration means that at each OR node, the latest alternative at or before the cutoff date is to be chosen. This corresponds to defining the newest revisions of all components at a given date.

**Default choices** One successor of each OR node may be designated as the default choice.

**Symbolic names** Symbolic names may be assigned to the branches of OR nodes. Thus, when choosing a configuration one could specify e.g., "VT100", and this would select the branch labeled "VT100" wherever there is a choice.
The introduction of cutoff dates is necessary and helpful. It may help the developer to identify and generate meaningful configurations based on the time of development of the individual configurations.

Even so the cutoff dates offer no help for the case when some alternative revision other than the one closest to the cutoff is to be included in the configuration. It then becomes necessary for the developer to combine cutoff dates with symbolic names and defaults. This is a lot of extra information to handle. Because AND/OR graphs for real size systems are large and detailed and provide no means of abstraction, the user must be extremely tidy not to lose track.

Another deficiency can be seen by observing the differences in the AND/OR graphs depicted in figure 4-15 and figure 4-17. The changes result from a revision of S where a new component was introduced. We observe a shift from representing S as an AND node with several components, where each of the components may have several revisions, to an OR node with alternative configuration versions each of which have several components. Thus, the result of introducing a new component C in S is a fundamentally changed AND/OR graph. This indicates that the system S now exists in two alternative versions each with possibly several revisions. In our opinion this does not adequately represent what really happened.
Furthermore, if we compare the AND/OR graphs in figure 4-13, figure 4-15, and figure 4-17 we see that the AND/OR graphs quickly grow large. The complexity of the graphs grows faster than the complexity of the system which they represent. This is in our opinion an undesirable property of a system which is meant to make it simpler to control growth of complexity during system evolution.

4.3. SCM for engineering databases

Version and software configuration control for engineering databases comprise systems which support a versioned data model to identify and control versions of complex product structures. Systems surveyed include Damokles, Change-oriented versioning, and Farandole 2.

4.3.1. Damokles

Damokles [39][40] was developed to give database support for the UNIBASE software engineering environment project in Germany.

The Damokles data model (DODM) is an Entity Relationship model extended with structured objects (complex objects) and long fields. An object is either a simple object or a structured object. A simple object is composed of a number of attributes, the descriptive part of the object, where one or more of the attributes serve as object key. Damokles automatically assigns a unique object identifier to an object at creation time. A structured object has a structural part, which is a set of subobjects that may be simple or structured.

The DODM also provides for relationships between objects and/or their versions. Relationships are n-place bidirectional associations of objects. Each place is characterised by a role attribute. Minimum and maximum cardinalities may be specified for each role. Relationships may be included in structured objects. Thus, all kinds of associations between and within objects may be defined.

Objects may have versions. The versions allow to represent multiple instances of the same object. Main characteristics of the version concept are:

- Versions are objects in their own right with a mandatory relation to exactly one object, its generic object. A version is identified by <object type name>.<version>. The <object type name> is the name of the generic object.
- The object attributes and the internal structure for a generic object are supposed to be common to all versions of the objects. However, the version attributes and the composition from subobjects may differ among the versions.
- Consequently, all versions of an object have the same kind of structure and the same attributes. There may even be versions of versions. Both the generic object and individual versions may be referenced.
- Versions of an object are numbered in creation sequence. Implicit predecessor-successor relationships are maintained among the versions. The structures may be linear, tree, or directed acyclic.
• Operators locate versions determined by the graph structure or by version number. It is possible to find the generic object of a given version. A version may be inserted or removed from the version graph of a generic object.

Damokles supports multiple databases, where a subdatabase belong to teams or to individual users. The following rules apply [40]:

• An object is a member of exactly one database. Copies are allowed in other databases.

• Relationships may be established between objects in different databases.

• All databases satisfy the same schema. However, the schema of an individual database may be just a part of the global schema.

Long design transactions are based on check-out/check-in of objects. A long transaction is started in the context of a subdatabase. The owner of a subdatabase is allowed to check-out objects from other databases. After completion of the work, the developer checks the objects back into the database. In the mean time others may still read the objects in the source database, but cannot check-out these objects themselves.

The versioning does not have any inherent connections with the long transaction mechanism. A check-out followed by a check-in does not necessarily create a new version. Hence, versions are not immutable, but behave as any other object with respect to updates.

Version handling is not transparent to the applications. An application which navigates via relationships in the database has two alternatives: Either a relationship leads to a specific version, or the relationship leads to a generic object, in which case the application itself has to navigate in the version structure to identify a unique version.

Software engineering in teams is supported by an extended access control facility. There are different lock modes for checked-out objects, depending on whether the checked-out copy is to be modified or read, and whether the write protection of the original object extends to its context. Complex consistency constraints are also included. They are based on events defined by the developer, which trigger actions defined by the developer.

4.3.2. Change-Oriented Versioning

The change-oriented versioning, COV, was introduced by Per Holager [67] and further elaborated by Anund Lie [98] and Bjørn Munch [111]. COV keeps version and configuration management orthogonal to the module structure. The concept of a functional change plays a major role. Versions, or bound configurations, are identified by a characteristic set of functional changes. Functional changes may apply to any number of modules, and COV accommodates independent changes to the same modules. Interdependencies between changes are recorded.

Options and validities

In the change-oriented model, the modifications that have been done to a development object is described by a set of options. An option is a Boolean
variable which is associated with a specific functional change. The value True for the option means that the functional change is going to be included. Logical relationships between options are called validities and determine valid combinations of options (functional changes).

Consider the following example which has been adopted from [98]:

A software product has been delivered for IBM-PC, Sun, and VAX computers, under the DOS, Unix, and VMS operating systems. The capability to run under X11 window system as an alternative to plain VT100 terminals, exist for the Sun version.

The following options exist:

IBM-PC, Sun, and VAX describe the processor.
DOS, Unix, and VMS describe the operating system.
X11 tells whether the X11 window system is in use.
UseMouse tell whether mouse support should be included

Example validities are

MutuallyExclusive(IBM-PC, Sun, VAX) Make sure that one and only one of those is selected
MutuallyExclusive(DOS, Unix, VMS) Make sure that one and only one of those is selected
DOS ⇔ IBM-PC Only DOS on IBM-PC and vice versa
VMS ⇒ VAX VMS only running on VAX
X11 ⇒ Sun ∧ UseMouse If X11, we must be on a Sun and mouse support must be included
IBM-PC ∨ VAX ⇒ ¬UseMouse No support for mouse for IBM-PC or VAX

If the developer wants to build a version for Sun with window support, the options Sun and X11 are chosen. Choosing these options explicitly, the validities implicitly determine that the following options are also chosen: Unix and UseMouse. As soon as a complete choice is made, it is possible to start accessing the database. The part of the database visible to the developer is the version corresponding to the option choice for each component.

Only choosing Sun, would not comprise a complete choice. If an incomplete choice has been made, it is still possible to access the database, but only those components which are not affected by the unspecified options may be accessed.

Long transactions

Adding a new feature or fixing an error, would amount to an update of the database. Updates are normally performed within the context of a long transaction. A long transaction in turn may start one or more edit tasks.

A long transaction is started by determining which part of the database one intends to work on, the ambition, during the task. A new option is added and the change is described. This includes adding validities which
relates the new option to the existing options, the task validity. An edit
task may start other edit tasks within the part of the database constrained
by the ambition of the parent task.

When the edit task terminates, the actual updates performed during the
task are merged back into the database, and the new validity of the soft-
ware product is determined as the set union of the old validity and the va-

dility of the edit task.

Transactions with overlapping ambitions are supported. The logical sub-
databases of each transaction are kept separate until the transactions fin-

ish. When there is overlap, the transactions involved are notified and ei-

der of the following may happen:

- The new transaction may wait for the older one to terminate, effec-
tively giving non-overlapping transactions.

- The transactions may negotiate which and how much of the ambi-
tions should be retracted.

- The transactions may continue unchanged, but are now being aware
  of that there is overlap among them.

When a transaction with an overlapping ambition terminates, all over-
lapped transactions are notified. A transaction receiving such notification
is expected to merge the updates committed by the first transaction into it-
sel. It will not be allowed to terminate before this has been done.

**Stability**

Versions are not immutable in COV. A modification does not produce a
new version to leave the old version unchanged. Instead, changes modify
existing versions. The feature that a particular choice will give the same
version now and sometime in the future is called *stability* of versions.
Stability is achieved by a boolean expression *stability* which defines which
choices are guaranteed to be stable. The *stability* formulae are formulated
by the system developer.

**Validity**

For the developer not every possible combination of options makes up a
"sensible" version. Comparatively few combinations of options of the total
option space are valid. To indicate inherent or intended restrictions on
combinations of options, COV has defined *validity* as a Boolean expression
determining those choices that produce valid versions.

Lie [98] identifies two different purposes of validities:

- Validities indicate inherent or intended restrictions on combinations
  of options.

- Validities indicate correctness or completeness of implementation
  for versions.

The first interpretation of validity focus on problem oriented, user defined
restrictions that apply to the options. A significant element of subjectivity
on the part of the developers comes into play when determining relevant
and "sensible" combinations of options.
The second interpretation of validity focuses on objective measures of correctness and completeness. Lie identifies a number of levels of increasing correctness: (1) No overlapping changes, (2) Syntactical correctness, (3) Static semantical correctness, (4) Simple module test successful, (5) Exhaustive module test successful, (6) Integration test successful, (7) Acceptance test successful.

The criteria above are tightly linked to the development language and development method used. They can therefore not be generally stated. Nor can they be automatically derived, since the informal nature of most contemporary methods and languages makes formulating and automatically generating such criteria impossible. Thus, the application of the latter interpretation of the validity criterion also relies on subjective evaluation on the part of the developers.

Configuration descriptions

COV is in many respects very different from the other version and configuration control models that have been presented in this chapter. Whereas the other models operate with object versions which themselves are explicit objects, COV is characterised by having version selection orthogonal to object selection. Versions are selected as the composition of optional sets of changes across all objects. In many ways this approach has much in common with Tichy's AND/OR model. The number of options may get very large. The complexity of expressing sensible validities increase very much as the number of potential combinations of options increases.

Although facilities for selecting mutually compatible versions of objects in a configuration is inherent in the change-oriented model, this happens only when developers consciously make it happen. Complex compatibility constraints need to be formulated to achieve this.

The COV-way of describing configurations has some undesirable properties. Firstly, the binary options lead to a very large state space from which the validities are determined. Secondly, there is no way to specify preferences, or to have the COV system complete a choice that is not complete. Thirdly, once a choice becomes stable is is impossible to return it to a non-stable status. Lastly, the validities are boolean. In general it is required that an option should be able to take more than two values, and their combinations into validities should accordingly be able to take more than two values.

Lie sketches a user oriented layer of configuration descriptions on top of the basic level of binary options with boolean connectives. The proposal is based on aggregates of options called features. Features have an enumerated set of possible values, each value representing mutually exclusive functionalities. Attributes are attached to object versions, and compatibility constraints are expressed by predicates over attributes. The attributes characterise functional properties of the objects across all objects, thus expressing inter object compatibility constraints.

Thus, the granularity of the level of control gets more coarse and complexity is accordingly reduced.
4.3.3. Farandole 2

The DBMS Farandole 2 was developed as part of the Rebirth project at the Université de Genève. The data model of Farandole 2 [3] may be considered as an extension of the Entity Relationship model. It is based on the concepts object, class, role, and generalisation/specialisation.

![Diagram of Farandole 2 data model]

Fig. 4-18: Schema of an airline company [3]

Modelling concepts

Objects of the same type are grouped into a class. There are two different types of classes: Atomic classes are terminal classes such as strings, integers, booleans, etc. Composite classes are aggregates of atomic classes.

A role has a name and a degree. It establishes a link between objects of two classes, an origin and a domain. The origin of a role is always a composite class. The domain may be either composite or atomic.

Objects of composite classes are represented by an identifier. The value of an object of a composite class is a tuple made up of objects linked to it by roles.

Generalisation and specialisation allows specialisation of a class into subclasses.

Farandole 2 introduced the concept of context to reduce the complexity of understanding a global schema, much like a user defined view. A semantic context is an abstraction which allows certain elements of a schema to be regrouped while others are masked.

Consider an example of a database schema developed for an airline company (figure 4-18). The example is adapted from [3]. The sub-classes of Person are Passenger and Staff. Sub-classes of Staff are Pilot and Radio, etc. A Flight is the association of Crew, Airplane, departure Airport, and arrival Airport. A Booking associates a Passenger with a Flight.
On the schema of figure 4-18 it is possible to define the context of Flight Planning, which is shown in figure 4-19.

The context may consists of the following nodes and edges:

**Nodes:** (p,Person), (st,Staff), (pi,Pilot), (r,Radio), (s,Student), (gr,Graduate), (c,Crew), (f,Flight), (a,Airplane), (a_dep,Airport), (a_ar,Airport).

**Edges:** {(c,Crew), (pi,Pilot)}, {(c,Crew), (gr,Graduate)}, {(c,Crew), (f,Flight)}, {(f,Flight), (a,Airplane)}, {(f,Flight), (a_dep,Airport), (air_dep)}, {(f,Flight), (a_ar,Airport), (air_arrival)}.

![Diagram of flight planning context](image)

Fig. 4-19: The flight planning context [3]

**Versioning**

Farandole 2 supports versioning of contexts. They have defined a version as follows:

*In this model, a version is defined as a stable and coherent state which the administrator or the designer desires to keep.*

Thus, the generation of a new version of a context is a process which results from a human decision. This means that all context modifications do not necessarily generate new versions.

From a context version it is possible to derive several versions. Additionally, a context version may be considered to be derived from several versions. Hence, the various versions form a directed graph.

There are two kinds of versions; working versions and stable versions. A stable version cannot be updated or deleted. Modifications are always performed on working versions. It is possible to move a working version into a stable version, and vice versa.
A user defined default version which is selected whenever a developer refers to a context without specifying a particular version, is supported.

A generic context means the set of versions of a context. It is described as follows:

[id, name, first_version, default, [working_versions],
[stable_versions], next_version, root_class]

_id is the internal identifier and name an external identifier of the context. First_version is the identifier of the first version of the version derivation graph. Default is the identifier of the default version of the context, while working_versions and stable_versions point to the working and stable versions respectively.

A version of a context is described as follows:

[id, gen_id, name, number, [successors], [previous], date, state,
[[nodes], [edges]]]

_Id is the internal identifier of the version. Gen_id is the identifier of its generic context. Name is an external identifier of the version. Number is a specific number which allows each version to be referenced. Successors is the set of its successors. Previous identified the set of versions from which it is derived. Date is the date of the last modification. State indicates whether the version is working or stable. Nodes and edges correspond to the lists of nodes and edges in the version.

**Working with context versions**

A schema consists of a set of contexts. Each context may exist in several versions. Versions of contexts are developed in one and the same database, from a common schema. This means that versions of contexts may share common elements. A modification performed on a version must therefore be clearly outlined with respect to other versions. Any modification carried out on a set of nodes and edges of a given version thus requires verification of whether they or their elements are present in other versions. Three rules have been defined to regulate possible operations on a version:

**R1** Adding an element E (node, edge, class, or role) to a version V constitutes either an addition or a creation, depending on whether E was or was not already defined in the schema. In the first case, E is integrated in V and is thus shared by several versions. In the second case, E must be created. The creation of an element is local to V. Addition of an element also requires application of the same procedure to its elements. For example, for a node, there must be verification that the associated class is already defined.

**R2** Suppressing an element E in a version V simply removes the element from V. Furthermore, after this, if E is not shared by another version and is isolated, then E is effectively deleted. This process is applied recursively to the elements which constitute E.

**R3** Modification of an element shared by other versions is done on a copy and has thus no effect on the other versions. For example, the modification, in a version V, of a class C figuring in several version is carried out on a copy of C.
To add a version of a context, either by creation or by derivation, a name must be given, a list of nodes and edges must be created, and it must be linked to the associated generic context. When the first version of a context is created, the root of the context must be chosen.

When a version is suppressed, all of its elements become inactive.

Using the rules R1, R2, and R3 as a bottom line, a number of operations on both the schema, the context structure, and contents may be defined. The semantics of the following operations have been defined [3]: Addition of a node, addition of an edge, suppression of a node, suppression of an edge, modification of a node, and modification of an edge. This includes the definition of a set of rules to guarantee the coherence of a schema after modifications have taken place.

The approach taken in Farandole 2 also takes into account the links between different versions of the schema and the objects stored in the database under the different schema versions. This has been realised by a set of mechanisms which allow the association of context versions with the objects pertinent to that version. The approach taken is able to identify that objects belong to a certain version, also considering objects that are shared among several versions, but it does not take into account the evolution of objects in a context version.

4.4. Support for SCM in commercially available CASE tools

A number of tools for information systems engineering have emerged in the marketplace. Some of these tools were surveyed in chapter 2. This section specifically looks into the support for version and configuration management provided by three selected contemporary CASE tools. The tools surveyed are Foundation by Andersen Consulting, IEF by Texas Instruments, and StP by Interactive Development Environments.

4.4.1. SCM in Foundation

Foundation is a set of loosely integrated tools originally developed by the Andersen Consulting to assist the company in providing information systems for its clients. Foundation is also commercially available.

Foundation supports the Method/1 development approach [4]. It includes three different life cycle models: One for custom system development, one for package system development, and one for iterative development.

The method includes the description of a large number of development tasks. Project management will in the start of the system development select the tasks estimated to be necessary for a successful project. It is also possible to manually specify project specific tasks and documents. Interdependencies among tasks are described by defining inputs and outputs for each task. Using this information, a project plan may be developed.

All development objects, including the conceptual models, are viewed as documents that are identified by type and name. Versioning of development objects is not supported automatically. Although a field on each
document indicates version, old versions are not kept. The developers are forced to use name conventions and copying of repositories to keep individual versions and configurations separate.

Each document has a status attribute associated. Additionally, the project management tool will keep track of the state of every task, based on information given by the developers.

On certain platforms and development environments, it is possible to generate parts of the application from the specification and designs developed by the design tools in Foundation [6]. In this case, the tool contains a systems building module and associated guidelines for what parts have to be rebuilt when other parts of the specification has been modified. Support for this functionality is semi-automatic. Since the tool does not generate complete code, the developer has to include business logic in the generated modules. It is possible to include developer's code in certain documents within the Foundation tool itself. This is usually not done, because it is difficult to know the exact content of the code until data structures etc. have been generated. Foundation helps in the development of *make* files for the system.

Foundation's design tool [5] supports working in a LAN environment, and uses a check-out/check-in strategy for access control of the development documents. The documents have referential links to other documents. It is possible to run checks to find references to non-existing documents.

A separate tool takes care of change requests [4]. Name referencing is used to decide which modules are influenced by a change. A change request has associated priority, status and other administrative details.

### 4.4.2. SCM in Information Engineering Facility

IEF was developed by Texas Instrument, originally for internal use [141][142]. As was described in section 2.5.1, IEF consists of four tool sets (section 2.5.1.1): Planning, analysis, design and construction (code generation).

IEF was designed to support the Information Engineering Method, IEM, developed by James Martin [108]. The development objects produced are matrices, documents and various diagrams.

Support for project management and process modelling is restricted to collecting specifications in the central repository called the Encyclopedia. The planning tool set is primarily used for business strategy planning tasks [65].

No support is offered for versioning of development objects. It is possible to support versioning management by using manual methods where the developer is responsible for keeping several copies of an existing system under different names. Development objects or aggregates of development objects may be migrated between the different modules, and the tool provides support for merging. There is no way to undo a migration that has already been performed.

IEF enables multiple developers to work on different parts of a model by splitting the model into several disjoint subsets. The process of splitting the model is called subsetting. Each submodel may then be further devel-
oped. Other developers may have read access to development objects that are not included in the subset they are working on. Problems with this approach has been reported [143]. It proves difficult to subdivide the database into disjoint subsets without getting into a situation where the same part of a model need to be accessible in two or more subsets. The subset checked out firstly, will then effectively block any other developer trying to check out the same elements in the model.

IEF supports generation of executable third generation language code (C or Cobol) from the diagrammatic specifications. The diagrams are tightly coupled. E.g., if the name of an attribute in an ER diagram is changed, the name of this attribute will change everywhere else in your model. Consistency is partly forced on the system by the tool during development through the tight coupling of the diagrams.

4.4.3. SCM in Software through Pictures

StP was developed by Interactive Development Environments, IDE [75][76]. While IEF and Foundation are normally considered to be ICASE tools providing life cycle support, StP is a point tool, supporting only parts of the development life cycle. It is a multi user CASE development environment that runs under Unix and VMS.

The StP environment is comprised of an integrated family of graphical editors and a document preparation system. Other development objects are code and code skeletons generated from the conceptual models. Project management issues are not covered by the tool set. Design rules to verify the completeness and consistency of diagrams exist.

StP runs on a heterogeneous network of workstations, minicomputers and X-terminals. It takes advantage of network based file systems and the X window system to provide users with transparent access to tools across the network.

StP includes the Version and Configuration Control, VCC, subsystem. The VCC is an interface to version control systems provided by the operating system platform. In a Unix environment the VCC interfaces either to SCCS or RCS. Under VMS, StP uses the Code Management System, CMS, and under the Domain/Aegis system it uses DSEE.

Versioning can apply to all project files or only some kind of project files in the database. When using the “native” version and configuration management tools through the interface provided by StP, only a limited set of the underlying functionality may be utilised. It is possible to bypass the VCC and use the native tools directly.

Concurrent access to the development database is controlled by a check-in/check-out mechanism. Files may also be checked out with read only access. When an updated version is checked in, the new version is stored, and the lock in the database is released. VCC also keeps track of development changes.
4.5. Chapter summary

This chapter has surveyed the version and configuration management field. Several approaches within the three categories file based systems, SCM for engineering databases, and SCM for CASE tools have been surveyed. We have identified that support for version and configuration management is weak in contemporary CASE tools. Advanced proposals for standalone approaches (the file based systems) and proposals for SCM integrated in engineering databases exist in the literature. The latter approaches are based on providing a versioned data model for complex product structures.

The appropriateness of these approaches to support information system modelling has not been demonstrated. Particularly the requirement for consistent evolving product structures introduces high complexity in the SCM systems while at the same time restricts cooperative work on the same structures since cooperative work necessarily means that the product structure is in a state of inconsistency most of the time. This also holds true for information system modelling in a CASE tool context. A notable exception is the Farandole 2 which in our opinion demonstrates adequate support for conceptual models [3].
5. Management of change

This chapter presents the functionality necessary to support the change process during information system engineering. This includes a framework for version and configuration management which makes it possible to model the various product structures involved in information system engineering, and which permits inconsistencies in the information system models. It also includes augmentation of the framework to support development transactions which allows asynchronous modes of working and coordination within the development team. Finally, it includes further augmentation of the framework for supporting synchronisation of the work performed by the individual members of the development team.

5.1. Version and configuration management

Version and configuration management keeps track of versions of system components developed by several developers working in a geographically distributed development environment. A version of a system is an immutable, identifiable edition of a system. A system is composed of a number of components. A component may either be a hierarchical composition of other (sub-)components or a flat structure with no hierarchical relationships among the (sub-)components. A version of a system is a composition of versions of system components.

The following issues are discussed: Definition of development object versions, definition of component structure, naming of components, and granularity of versioned components. The approach is introduced through an example.

5.1.1. Development object versions, an example

Consider the activity of replenishing stock. The activity is initiated (triggered) when a stock clerk reports that the amount in stock of a part, is lower than a threshold value. Existing sales orders will have to be compared with historical sales figures in order to give the basis for determining the quantity to buy. The supplier who can give the best offer for delivering the part will be awarded the order.

Figures 5-1, 5-2, 5-3, 5-4 and 5-5 show five different versions of the Data Flow model of the Replenish Stock activity. Versions are given unique names based on the name of the development object followed by a version number, e.g., Replenish Stock 1.0. Versions which are meant to exist in parallel, variants, are separated by adding a variant specific name in brackets just after the version number, e.g., Replenish Stock 1.2[Monopoly].

We assume that the system model has been refined from a first approximation (version 1.0) consisting of only two processes Receive Order and Place Order (figure 5-1), via a version 1.1 (figure 5-2), into two different variants (version 1.2 and version 1.2[Monopoly]). The two variants may have been developed in order to serve two different markets with different market characteristics, and may thus exist side by side. Version 1.2 is a
revision of version 1.1 and is thus meant to replace version 1.1. Observe that the processes as well as the data stores have changed their names as the model has developed.

The initial version, 1.0, is depicted in figure 5-1.

Fig. 5-1: Replenish Stock 1.0

Replenish Stock 1.0 is revised and version 1.1 is produced (figure 5-2). Version 1.1 is based on, and is a revision of, version 1.0. Process Receive Order (in version 1.0) has been deleted and replaced (in version 1.1) by the two processes Register Need for Part and Find Best Supplier. The number of a process is an alias for the process name, local to the development object in which it exists. Accordingly, the numbering of the processes in version 1.1 has changed to reflect the structure of the development object.

Fig. 5-2: Replenish Stock 1.1

Suppose that the Replenish Stock Data Flow model in figure 5-2 was developed for markets with many suppliers. To handle markets where there is only one supplier the model needs to be modified. Figure 5-3 depicts version 1.1[Monopoly]. The process necessary to find the best supplier (in version 1.1) is not needed and has been deleted. Version 1.1[Monopoly] is based on version 1.1 and will exist in parallel with version 1.1 as a variant.

Fig. 5-3: Replenish Stock 1.1[Monopoly]
Process *Register Need for Part* in version 1.1 does not take into account the processing necessary to estimate the quantity one wants to order. To improve this situation the process *Register Need for Part* (in version 1.1) has been replaced by *Estimate Future Need* to create version 1.2 (figure 5-4).

![Diagram](image)

**Fig. 5-4: Replenish Stock 1.2**

A parallel revision to the one which took place from version 1.1 to version 1.2, to reflect the need to estimate the order quantity, has also been made to version 1.1 [Monopoly]. Figure 5-5 depicts version 1.2 [Monopoly] where *Register Need for Part* (version 1.1 [Monopoly]) has been replaced by *Estimate Future Need*. Version 1.2 [Monopoly] replaces version 1.1 [Monopoly].

![Diagram](image)

**Fig. 5-5: Replenish Stock 1.2 [Monopoly]**

### 5.1.2. Definition of development object versions

Each development object may exist in a number of versions. There are two classes of versions: *Revisions* are versions that are meant to replace previous versions, and *variants* are versions that are meant to coexist with previous versions. Two versions can thus be either revision related or variant related.
The difference between two succeeding versions can be characterised by a set of elementary (atomic) changes. We take the assumption that each version is developed based on a unique existing version. The default initial version is 1.0. E.g., version 1.1 of Replenish Stock was developed based on version 1.0, version 1.2 was developed based on version 1.1, etc..

A version of a development object is named as follows:

development-object-name revision-number[variant-name],

where development-object-name identifies the development object, revision-number identifies the revision and is comprised of two numbers separated by a ".", and variant-name identifies the variant chain to which the version belongs, e.g., Replenish Stock 1.2[Monopoly].

Revisions are given a two digit revision identifier in addition to their name, e.g., Replenish Stock 1.1. The two digits make it possible to distinguish between major and minor version changes. Variants are given a variant specific name in addition to development object name and revision identifier, e.g., Replenish Stock 1.1[Monopoly]. The name Monopoly in brackets identify this version as variant related to version 1.1. The semantic difference between version 1.1 and version 1.1[Monopoly], is that the latter version was developed for markets where there is only one supplier in the market, while the former was developed for a multi supplier market.

![Diagram](image_url)

Fig. 5-6: Version graph for Replenish Stock

We propose the version graphs to graphically depict the relationships between versions. Figure 5-6, depicts the version graph for the Replenish Stock example. The development object Replenish Stock exist in five different versions. The initial version is version 1.0. Version 1.1 was developed based on version 1.0. Version 1.2 was developed based on version 1.1. Version 1.1[Monopoly] was developed based on version 1.1 to serve markets where there is only one supplier in the market.

Figure 5-7, depicts a version graph for a generic system $S$, which exist in ten different versions. The initial version is $S\ 1.0$. Two variants, called $S\ 1.0[AA]$ and $S\ 1.0[BB]$ respectively, have been developed based on $S\ 1.0$. Further, version $S\ 1.1$ has been developed as a revision of $S\ 1.0$. 
But for the precedence relationships indicated by the arrows between versions, the graphical representation in the version graph contains no information on time. I.e., it can be seen from the graph that \( S\ 1.0^{[AA]} \) was developed after \( S\ 1.0 \), but it is not possible to tell whether \( S\ 1.0^{[AA]} \) was developed before or after say \( S\ 1.1 \). Detailed information on creation date and time is associated with each version.

![Version graph for system S](image)

Fig. 5-7: Version graph for system \( S \)

### 5.1.3. Definition of component structure

A system comprises a number of system components. The components are either a hierarchical composition of other (sub-)components or a flat structure with no inherent hierarchy. An example of a hierarchical component structure is a set of Data Flow diagrams which are hierarchically ordered by way of decomposition relationships from processes in one diagram to other (sub-)diagrams. An example of a flat component structure is an Entity Relationship diagram where there are no hierarchical relationships among the different components of the diagram. The two types of component structures require different treatment with respect to versioning. Thus, we make the distinction between hierarchical and flat models.

**Hierarchical component structures**

The hierarchical models are characterised by hierarchical decomposition. Decomposition of a component is a refinement of the component in terms of a more detailed structure of lower level, i.e. more detailed, components. Hence, every system component is either atomic or compound. An atomic component has not been further decomposed, while a compound component has been further decomposed into a set of system components. Every system component exists in a number of versions. Thus, a specific version of a non atomic component is composed of specific versions of its subordinate components.
In a standard hierarchy a modification in a leaf node component, which will create a new version of that component, would result in a ripple of version changes all the way to the top of the hierarchy.

Consider the hierarchy depicted in figure 5-8. A new version of system component $J$ will result in a new version of $E$, which in turn will result in a new version of $B$, and finally a new version of $A$. Similarly, a new version of $K$ will result in yet another version of $E$, which in turn will result in a new version of $B$, and finally another version of $A$. Thus, both $E$, $B$, and $A$ have changed twice due to changes in $J$ and $K$ respectively. If $J$ and $K$ had been modules in a software system, the system $A$, would have changed twice. This may be desirable. It may also be that the two modifications that are made to $J$ and $K$ were meant to be done in one version change. This ripple of version changes is in many cases an undesirable property of the hierarchy which leads to difficulties in handling temporary inconsistencies. A number of small modifications in components deep down in the hierarchy will result in a large number of version changes for components higher up in the hierarchy. Thus, automatic creation of a new version of a component whenever there is a version change in a subordinate component is undesirable. We propose to let system components be related to their decompositions by a solved by relation (bold face arrow) to avoid this cascading of changes of versions [156].

![General hierarchy](image)

Fig. 5-8: General hierarchy

If we use the solved by mechanism it is possible to delay modifications to a component (i.e. new versions,) to take effect on components higher up in the hierarchy until they are explicitly included. Thus, new versions of components are created explicitly. Once a new version of a component becomes available, it is not mandatory to include it in any superior component. The inclusion of any new versions of subordinate components is done explicitly. Hence, inconsistencies among the development objects are tolerated.

We propose the component structure graphs to represent system structure, where each component may exist in a number of versions, and the solved by relation represent the relation between a system component and its subordinate components. Figure 5-9 depicts the general hierarchy from figure 5-8 as a component structure graph. System component $A$ consists of system components $B$, $C$, and $D$. Component $B$ is related to the
decomposition of $B$ by the solved by relation. This is also indicated by giving component $B$ the name $A.B$, thus indicating that it acts as a pointer to the decomposition of the component.

![Component structure graph](image)

Fig. 5-9: Component structure graph

The component structure graph for version 1.1 of Replenish Stock (from figure 5-2) is depicted in figure 5-10. Version 1.1 of Replenish Stock contains five components. The parenthesis enclosing the version number for the components indicate that this is really a solved by pointer to the appropriate version of the components.

![Component structure graph for Replenish Stock 1.1](image)

Fig. 5-10: Component structure graph for Replenish Stock 1.1
There are four possible operations on hierarchical components: *New hierarchical, modify hierarchical, remove hierarchical, and merge hierarchical.*

*New hierarchical* defines a new hierarchical component from scratch. The new hierarchical operation defines a placeholder in a new version graph for the new hierarchical system component. The default initial version is 1.0.

*Modify hierarchical* defines a new version of a hierarchical component. The old version of the hierarchical component is left unchanged. The new version may be either a revision or a variant of the previous version. The successions of versions are handled by the version graphs proposed in the previous section.

*Remove hierarchical* deletes the last version of a hierarchical component. Versions of a hierarchical component within a revision chain cannot be removed. The remove hierarchical operation does not affect any other versions of the hierarchical component.

*Merge hierarchical* merges two hierarchical components into a single component. The merge hierarchical operation takes as input two different development versions of a hierarchical component belonging to two different development transactions. The operation creates a new hierarchical component which contains the result of the merging operation. The merging operation may be entirely manual or semi-automatic.

When hierarchical components are merged, names of the sub-components existing in the different hierarchical components are compared. Names of sub-components in the hierarchical components that are being compared, the "input" components, are left unchanged. In the case of a conflict of names among the "input" components in the merging process, the sub-component in question will have to change its name in the merged component. Name conflicts will in general have to be resolved manually.

![Diagram](image-url)

Fig. 5-11: *Estimate Future Need 1.0*
The example revisited

Process P1, *Estimate Future Need*, and process P2, *Find Best Supplier*, in version 1.2 of *Replenish Stock* in figure 5-4 may be decomposed. Figure 5-11 depicts a decomposition of process P1, *Estimate Future Need 1.0*.

Figure 5-12 depicts a decomposition of process P2, *Find Best Supplier 1.1*.

As shown in the version graph in figure 5-6, *Replenish Stock* exist in five different versions. As can be seen from comparing figure 5-2 with figure 5-4, a number of modifications have taken place from version 1.1 to version 1.2: Two of the processes are different, a new data store has been introduced, and two of the processes have been decomposed (figure 5-11, 5-12). The component structure graph has changed accordingly.

In figure 5-13 the component structure graph for version 1.2 of *Replenish Stock* is depicted. The component structure graphs for *Estimate Future Need 1.0* and *Find Best Supplier 1.1* are connected by solved by relations.

The component structures depicted in figure 5-10 and 5-13 show the external view of the system structure. This external view is really the view of what is often called a system configuration, i.e. a specific choice of compatible versions of all components on all levels in the system hierarchy. The system configuration is a specific selection in the total picture comprised by the version graphs coupled to the set of system component graphs connected by solved by relations. This total picture, called the storage view, is illustrated in figure 5-14.

---

**Fig. 5-12: Find Best Supplier 1.1**

Flat component structures

The flat models, e.g., Entity Relationship diagrams, have no inherent hierarchical structure. Components of varying significance and multiple levels of detail are logically placed on the same level of abstraction, i.e. in the same diagram. E.g., entities and attributes, entities and specialised
entities, entities and relationships, would all be depicted in the same Entity Relationship diagram. There is generally no concept of decomposition, e.g., entities are seldom decomposed. Notable exceptions exist: The ERT language, an Entity Relationship dialect defined in the context of the Tempora Esprit project [19], has proposed the notion of complex entity types. The complex entity found in the ERT of Tempora has a graphic notation to simplify the diagrams in addition to specific semantics with respect to the representation of the model for e.g., insertion and deletion. However, this does not mean that decomposition in the sense used for hierarchical models exists in flat models.

Generally, every system component that is of the flat model type is atomic in the sense that it is not further described by any decomposition. Every system component of the flat type will nevertheless be subject to development and subsequent modifications and thus exist in a number of versions. Although system components of the flat type are considered atomic from a system structure perspective they are not atomic from a systems development perspective. E.g., a process in a Data Flow diagram will only reference certain entities, attributes, and relationships in an Entity Relationship diagram. A subdivision of tasks among several developers based on the hierarchies in for example Data Flow diagrams, will thus have as a consequence that all of the tasks reference, and may want to modify, the same flat component, e.g., the Entity Relationship diagram. Consequently, there is a need for developers to be able to reference, enhance, and modify only parts of a flat model, although this flat model is common to a large number of developers.

![Component structure graph for Replenish Stock 1.2](image)

Fig. 5-13: Component structure graph for Replenish Stock 1.2
We propose to use the concept of *scenario* [137][139][138], to impose a structure on top of flat components. In its original definition the *scenario* is defined relative to a real world phenomenon model proposed by Sølvberg [136]. A *scenario* is defined as follows:

"A *scenario* is a collection of phenomena classes".

"A phenomenon class is a subset of phenomena within the universe of discourse".

"A phenomenon is something as it appears in the mind."

Different application projects will develop different scenarios based on differences in application needs and perception of real world phenomena. Hence, the scenarios constitute a classification of real world features according to the needs and perceptions of the applications. The scenarios may therefore be used to facilitate co-existence and integration of discourse systems developed by different application projects. The union of the scenarios constitute the complete conceptual model.

Fig. 5-14: Storage view of the version and component structure graphs

In our opinion the word *scenario* has a dynamic connotation which is in conflict with the static nature of phenomenon models. In this thesis we
therefore decide to use the concept *conceptual view* as analogous to the *scenario* concept defined by Sølvberg.

The concept of *conceptual views* may be generalised and applied to all flat models. In our context the definition of a *conceptual view* becomes:

"A conceptual view is a logically connected subset of a flat model".

Conceptual views may overlap, such that the same development component may exist in several conceptual views. There are no "live links" between components occurring in several conceptual views, e.g., updates are not automatically propagated among conceptual views. Conceptual views may exist in several versions.

As mentioned above, one of the prime purposes of introducing the conceptual view is to facilitate co-existence of discourse systems. This means that conceptual views are related, and thus form a structure. A conceptual view can be built by selecting components from an existing conceptual view. When a conceptual view is built by selecting components from an existing conceptual view, the relationship, *built from*, between the existing and the new conceptual view is recorded. The default initial version of the new conceptual view thus created is 1.0. The *built from* relationship may be applied on all conceptual views, thus creating hierarchies of conceptual views built from conceptual views built from conceptual views, etc..

There are four possible operations on a conceptual view: *New conceptual view*, *modify conceptual view*, *remove conceptual view*, and *merge conceptual view*.

*New conceptual view* defines a new conceptual view from scratch. The new conceptual view operation defines a placeholder in a new version graph for the new conceptual view. When a conceptual view is built from scratch, no indication of origin is given. The default initial version of a conceptual view is 1.0.

*Modify conceptual view* defines a new version of a conceptual view. The old version of the conceptual view is left unchanged. The new version may be either a revision or a variant of the previous version. The successions of versions are handled by the version graphs introduced in the previous section.

The concept of variant of a conceptual view would be used to capture situations where it is desirable that different versions of the same conceptual view co-exist over time. For example, an Entity Relationship diagram that exists in two versions that are variant related, to serve as the basis for two variant related Data Flow diagrams. An alternative way of realising parallel versions is to build a new conceptual view based on an existing version. This would establish the new conceptual view as *built from* related to the original conceptual view, and not as a variant to its origin. The differences are few. But, rather than restricting the use of the variant graphs applied to conceptual views to simple revision chains, we have decided to keep the "redundancy" and leave the choice between the two alternatives to the preferences of user.

*Remove conceptual view* deletes the last version of a conceptual view. Versions of conceptual views within a revision chain cannot be removed.
The remove conceptual view operation does not affect any other conceptual views or any other versions of the conceptual view.

*Merge conceptual view* merges several conceptual views into one single conceptual view. The merge conceptual view operation retains the old conceptual views, and creates a new conceptual view which contains the result of the merging operation. The merging operation may be manual or semi-automatic.

When conceptual views are merged, names of concepts existing in the different conceptual views are compared. Names of concepts in the conceptual views that are being merged, the “input” conceptual views, are left unchanged. In the case of a conflict of names among the “input” conceptual views in the merging process, the concept in question will have to change its name in the “merged” conceptual view. In any case the relationship between the name of a concept in the input conceptual view, and the name of the same concept in the merged conceptual view, is kept.

There are four different possibilities:

- Identical names have been used for identical concepts. Name uniqueness is preserved without changes.
- Identical names have been used for different concepts. One of the concepts will have to be renamed to ensure name uniqueness within the new conceptual view.
- Different names have been used for different concepts. Name uniqueness is preserved without changes.
- Different names have been used for identical concepts. One of the names will be kept, and the other identical components will be renamed accordingly to ensure name uniqueness within the new conceptual view.

In general, name conflicts will have to be resolved manually.

The example revisited

An Entity Relationship diagram for the *Replenish Stock* activity is depicted in figure 5-15. The component is called *Stock Handling*. The model depicts concepts (entities, attributes, and relationships) relevant to the *Replenish Stock* activity. This is an example of a flat model. There is no deep conceptual difference between the concepts modelled as far as abstraction level is concerned. E.g., the concept of an *Order*, the *Price* (attribute of a *Part from Supplier*), and the *Supplies* relationship between *Supplier* and *Part from Supplier* are all conceptually on the same level of abstraction. The reasons for modelling e.g., the concept *Order* as an entity and not as a relationship, are purely pragmatic and dependent on features of the application and features of the modelling language.

The model in figure 5-15 covers most of the concepts used in version 1.2 of the *Replenish Stock* activity as it is modelled in figure 5-4. Suppose that further development is split between two teams of developers, the *Need Figures* team, and the *Suppliers* team. The *Need Figures* team is assigned the responsibility of process *Estimate Future Need* and the *Suppliers* team is assigned the responsibility of processes *Find Best Supplier* and *Place Order* respectively. An analysis of the in-
formation requirements of the Estimate Future Need process will show that only a subset of the concepts depicted in figure 5-15 are needed. An analogous analysis can be performed for the Find Best Supplier and Place Order processes. The result would be that the information requirements of the processes assigned to the two teams are different but overlapping. To minimise dependencies between the teams two conceptual views are built. The conceptual views, called Need Figures and Suppliers after the teams, are depicted in figures 5-16 and 5-17 respectively. The teams may now work for a while, without the added costs of having to co-ordinate their efforts.

![ER Diagram](image)

Fig. 5-15: ER diagram, Stock Handling, version 1.0

The conceptual views are now allowed to develop independently. After some development within the teams the conceptual views may be inconsistent. The concepts originally present in both conceptual views, e.g., Order and Date of order, may be changed in any of the conceptual views. When the work done by the two teams are going to be reconciled, the two conceptual views will have to be merged.

Suppose now that the Need Figures team modifies the Need Figures conceptual view. The modification consists of adding an attribute, Date of delivery to the Order entity. The modification is motivated by a requirement to check back orders. The requirement was uncovered in the decomposition of process P1, Estimate Future Need, version 1.0, depicted
in figure 5-11. The modified conceptual view, version 1.1, is depicted in figure 5-18.

Further inspection of the decomposition of process P1, Estimate Future Need, version 1.0, depicted in figure 5-11, leads to another modification of the Need Figures conceptual view. The modification is motivated by the need to improve the basis for calculating present need figures. The modification consists of adding entities Marketing Dept. and Production Plans, and the relationships Produce with attributes Date and Qty, from Production Dept. to Product, Sell with attributes Date and Qty, from Marketing Dept. to Product, Influence from Marketing Dept. to Production Plans, and Use from Production Dept. to Production Plans. Version 1.2 of the Need Figures conceptual view is depicted in figure 5-19.

Fig. 5-16: Conceptual view Need Figures, version 1.0

Fig. 5-17: Conceptual view Suppliers, version 1.0
Motivated by the need to analyse suppliers uncovered in the decomposition of process P2, *Find Best Supplier*, version 1.1, depicted in figure 5-12, the *Suppliers* team makes a modification to the *Suppliers* conceptual view. The modification consists of adding an attribute $QoH$ to entity *Part from Supplier*, and adding attributes *Sales Figures* and *Revenues* to entity *Supplier*. Version 1.1 of conceptual view *Suppliers* is depicted in figure 5-20.

The conceptual views *Need Figures* and *Suppliers* has been allowed to develop independently. The two teams decide to work out a reconciled version of the conceptual views at this point in time. The last version of the
conceptual views which originally were built from the same model (conceptual view *Stock Handling 1.0* in figure 5-15) are at this stage inconsistent. The concepts originally present in both conceptual views have been changed in one of the conceptual views. Successful merging of the two conceptual views depends in this case on recognising the fact that even though the entity *Order* has two attributes in the *Suppliers* conceptual view and three attributes in the *Need Figures* conceptual view, it is indeed the same concept being modelled in both conceptual views. Thus, the entity *Order* will have three attributes in the merged version. The attributes are *Date of order* and *Qty* which are common in both “input” conceptual views, and *Date of delivery* which only exists in the *Need Figures* conceptual view. Manual inspection shows that the remaining concepts in the two “input” conceptual views are either overlapping and identical, or not overlapping (neither in intention nor extension).

In the general case, manual intervention to negotiate the inconsistencies that have developed among the “input” conceptual views is required. Figure 5-21 depicts the result of the *merge conceptual view* operation, *Stock Handling*, version 1.1. This also indicates that the result of the merging has been logically related to the original conceptual view *Stock Handling 1.0* as a revision.

![Conceptual view suppliers](image)

*Fig. 5-20: Conceptual view *Suppliers*, version 1.1*

The development situation just described is depicted in figure 5-22. The figure shows that the two conceptual views *Need Figures* and *Suppliers* both emanate from version 1.0 of conceptual view *Stock Handling*. The *Need Figures* conceptual view is further developed, producing two consecutive revisions, version 1.1 and version 1.2. The *Suppliers* conceptual view is developed into version 1.1. At this point in the development, a merging of conceptual views is performed. The “input” conceptual views of the merge operation are *Need Figures 1.2* and *Suppliers 1.1*. The result of the merge operation is a “merged” conceptual view. The “merged” conceptual view is by default suggested to be a revision of the conceptual view from which (one of) the “input” conceptual views of the merge operation was built, i.e. in this example a revision of *Stock Handling 1.0*. The “merged” conceptual view may be logically related to any other version of a conceptual view, or left as a standalone version of a conceptual view.
Fig. 5-21: Conceptual view *Stock Handling*, version 1.1

Note that it is generally not possible to automatically determine any "optimum" conceptual view to which the "merged" conceptual view by default would be related. The definition of "optimum" is subject to personal interpretation and the intention of the merge operation rather than the origin and history of the "input" conceptual views.

Fig. 5-22: Development of conceptual views at time=t₁
Following the merge operation that produced the reconciled conceptual view, Stock Handling 1.1, the Suppliers team and the Need Figures team continue their individual work. Further development in either of the teams is best served by allowing the development to continue from where it stopped before the merge operation. In this case, further work in the teams is best based on the two conceptual views Suppliers 1.1 and Need Figures 1.2, respectively. To base further work on the original "input" conceptual views, has the benefit of allowing the teams to capitalise on previous work and stay within the discourse system that originally was developed by the team. Remember that the relationship between concepts that are "identical" in the "input" conceptual views and the "merged" conceptual view is kept, such that it is still possible to relate these concepts in subsequent versions of the Suppliers and Need Figures conceptual views to the same concepts in the previous version of the "merged" conceptual view.

Fig. 5-23: Development of conceptual views at time=t2

It may be the case that the result of the merge operation is such a major restructuring of either of the "input" conceptual views, that the feeling of the teams is that it would be counter-productive to continue to use the
“input” conceptual views as the basis for further development. This may be particularly so knowing that a new merge operation inevitably is going to take place at some future point in time. In this case, an alternative solution is to build two new conceptual views based on the reconciled version, and use these new conceptual views as the basis for further work.

In our example the two teams decide to continue their work based on the original “input” conceptual views, i.e. Suppliers 1.1 and Need Figures 1.2. The continued development is depicted in figure 5-23.

Development in the Need Figures team produces a new version, Need Figures 1.3. Development in the Suppliers team produces two new versions of conceptual view Suppliers, 1.2 and 1.3 respectively.

Simultaneously, the Suppliers team has had a development track seeking to build an Entity Relationship diagram for the special case of Norwegian suppliers. This development track was started with a conceptual view, Norwegian Suppliers 1.0, that was built from Suppliers 1.1. The Norwegian suppliers track produces versions 1.1 and 1.2, respectively. The Suppliers team then decides to merge the results of the Norwegian suppliers track, Norwegian Suppliers 1.2, with version 1.3 of the Suppliers conceptual view. The result of the merge operation is determined to be a new revision of the Suppliers conceptual view, Suppliers 1.4. Note the fact that Suppliers 1.4 is both a revision of Suppliers 1.3 and the result of a merge operation where the “input” conceptual views are Suppliers 1.3 and Norwegian Suppliers 1.2.

Eventually, the Suppliers and the Need Figures teams again merge their conceptual views, and the result is called Stock Handling 1.2, a revision of version 1.1 of the original Stock Handling conceptual view.

5.1.4. Global name uniqueness

It must be possible to refer to all development objects in an information system by a unique name. We will present a naming schema for development objects that provide and maintain global name uniqueness.

There are two alternatives for obtaining global name uniqueness:

- A flat namespace where the name uniquely identifies the development object no matter where it is being used.
- A hierarchical namespace where development objects’ names are qualified with the names of hierarchically superior development objects.

The simplest solution comprises a flat namespace where a development object’s name uniquely identifies the development object independently of where it is used. For small systems this is an ideal solution. Each development object has its own name distinct from any other development object. Referring to a development object by name is simple and without ambiguity. However, as the number of development objects grow large a flat namespace will quickly become impossible to manage.

A solution where development objects’ names are qualified with the names of hierarchically superior development objects remedies the prob-
lem of namespaces growing out of bounds. A hierarchical qualification system allows a flat namespace of moderate size to be seen as a hierarchy.

We propose to have a hierarchy of project libraries. Each development object should be uniquely named within each library. Development objects within a development object should be uniquely named within that development object. Global uniqueness is accordingly ensured through qualification of a development object’s name with the name of the superior development object.

Hierarchical models

In the case of hierarchical models, the name uniqueness requirement imply that each development object will have to be uniquely named within each development object for each library. Names of components are qualified by the name of the ancestor development objects. Thus, in the case of e.g., Data Flow diagrams, the name of each component in a diagram will have to be unique. Each process, data store, data flow, and external entity is uniquely identified by its name within a diagram.

The process numbers and data store numbers in Data Flow diagrams act as alias names for processes, and data stores. The scope of the alias is local to the diagram. The fact that the numbering of processes or data stores may change from one version to the next is therefore of no significance as far as the identification of the individual processes or data stores is concerned. It merely reflects a change in the total number of processes or data stores present in the diagram. E.g., process P3, Place Order, in Replenish Stock 1.1 changes its number to P2, Place Order, in Replenish Stock 1.1[Monopoly] due to a change in the total number of processes.

Flat models

Flat specification models inherently have a global name space. The conceptual view concept superimposes a structure on the models. The structure is motivated purely by pragmatic considerations within the development organisation, e.g., the split of tasks among developers. The name uniqueness requirement applied to conceptual views imply that components’ names will have to be unique within a conceptual view and not within the complete conceptual model for a domain. Names in different conceptual views are not necessarily unique. Names of components are qualified by conceptual view name. E.g., in the case of a conceptual view defined over an Entity Relationship diagram, each component in the conceptual view will have to be unique. Each entity, relation, and attribute, is uniquely identified by its name within the conceptual view.

5.1.5. Granularity of versioned components

To be able to keep concurrent versions of system components, we must determine a suitable level of granularity of atomic system components. That is, we must decide on the atomicity of system components. The granularity will have to be determined by the grouping of selected basic modelling constructs into appropriate atomic components. The atomic system com-
ponents will subsequently be composed into structural components which will form a hierarchy of system components.

The decision on granularity will reflect a compromise between the high complexity and high costs that are associated with a fine grained solution, and the relative loss of control and flexibility that come with a coarse grained solution.

Hierarchical models

As a representative of hierarchical models, we consider the modelling constructs of Data Flow diagrams. The Data Flow diagrams comprise a natural hierarchy through the decomposition of processes. Revision of a process happens through a revision of its decomposition. Other components of Data Flow diagrams are data stores, data flows and external entities. Those components are not refined through hierarchical decomposition as is the case for processes. From a version handling perspective, refining a data store or a data flow is equivalent to deleting the old data store or data flow and inserting a new. Thus, the process component of a Data Flow diagram is the lowest level development object that will exist in several versions.

Flat models

Regarding flat models, e.g., Entity Relationship diagrams, the feasibility of keeping versions of entities, relationships, and attributes is in our opinion questionable. The feasibility of keeping versions of conceptual views, i.e. "entire" diagrams of some manageable size, is much more obvious.

5.1.6. Selecting a version

A specific version of a development object may be selected and accessed by browsing of the component graph in combination with simultaneous browsing of the version graph.

- Having chosen one specific development object version in the component structure graph it is possible to switch to the corresponding version graph to see the complete version picture for that development object.

- Having chosen one specific version of the development object in the version graph it is possible to switch to the corresponding component structure graph to display its "subordinate" components. If this is an atomic component one may switch to browsing of its content.

- Having chosen one specific version of a development object in the version graph it is possible to start the development of a new version of the development object based on the selected version. The new version can be related as serial or parallel to its origin.
5.2. Way of working

This section briefly describes a development conceptual view which serves as an illustration of the use of the concepts and functionality presented in the next two sections. The section describes the application of these concepts to the PPP, an integrated CASE tool [62]. Parts of this proposal have already been implemented [153].

Selecting a version of a development object

A specific version of a development object is selected and accessed through browsing of the component graph in combination with browsing of the version graph as indicated in the previous section.

For example selecting version 1.2 of *Replenish Stock*, is done by marking it in the version graph in figure 5-6, and choosing browsing or check-out from a pop-up menu. Browsing would result in the creation of a read-only window displaying the contents of the development object using the appropriate tool or editor. Check-out would result in the creation of a placeholder in the version graph. The placeholder will have to be placed either as a revision or as a variant to the version that was checked out.

Development transactions

When a version of a development object is checked out for further development, the version of the object is copied to a private workspace and may be modified by the developer using a set of appropriate tools. Elapsed time before check-in may be in the order of days and weeks. When the modifications are completed the object is checked in to the specification database. The task of developing a new version of a development object, i.e. from check-out till check-in, is termed a development transaction.

As development transactions may be arbitrarily long lived, facilities for sharing of specifications between transactions are provided. Several transactions are permitted to concurrently access and check-out the same version of a development object. Thus several copies of the same version may be in the process of being modified simultaneously by different transactions.

Modifications made to the development object by the different transactions are by default considered private and kept separate until check-in time. Synchronisation among transactions is provided by the publish/subscribe mechanism that allows one transaction to subscribe to local revisions of development objects developed and published by other transactions.

Check-in time is determined by the developer responsible for the development object.

Suppose that two different developers were interested in modifying *Replenish Stock* 1.2 with the intention of creating a new version. Both developers would check out the development object in separate development transactions, say *Trans-1* and *Trans-2*. For each transaction initiated a transaction log is created. The transaction log is owned by a transaction, and contains information on the development objects currently checked out by the transaction.
When the first transaction checks out the development object, a place-
holder for the new version is created and a check-out contract for the new
version of the object (the placeholder) is created. The check-out contract
will be updated with information on date and time of check-out, transac-
tion identifier, whether local revisions created during the transaction are
to be published or not, and subscriptions to local revisions of development
objects created by other transactions. Let us assume that the first transac-
tion checking out version 1.2 of Replenish Stock places the new place-
holder as a revision of version 1.2. The placeholder will thus be named
Replenish Stock 1.3. The check-out contract for Replenish Stock 1.3, after
transactions Trans-1 and Trans-2 have checked out the object is depicted
in figure 5-24.

<table>
<thead>
<tr>
<th>Check-out Contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object id: Replenish Stock 1.3</td>
</tr>
<tr>
<td>Status: Checked out</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Transaction id: Trans-1</td>
</tr>
<tr>
<td>Date: 06.01.92 : 09:45:20</td>
</tr>
<tr>
<td>Publish: Yes</td>
</tr>
<tr>
<td>Subscribe: None</td>
</tr>
<tr>
<td>Transaction id: Trans-2</td>
</tr>
<tr>
<td>Date: 08.01.92 : 12:05:10</td>
</tr>
<tr>
<td>Publish: No</td>
</tr>
<tr>
<td>Subscribe: (Object id = Replenish Stock 1.2, Transaction id = Trans-1)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Fig. 5-24: Check-out contract for Replenish Stock 1.3

Check-in of a new version of a development object

At check-in time the development object is returned to the specification
database with the purpose of updating the database with a new version of
the development object. If there are several candidates for the new ver-
sion, i.e. if the development object was checked out by several transactions
and these transactions have modified the object, the candidate versions
must be merged. All developers having a copy of the development object
are notified, and a suitable time to attempt the merging is negotiated.
Some developers may need an extra day or an extra week before being
ready to merge with the others. In the end, the time of merging will have
to be determined by the developer formally responsible for the development
object.

When the merging of the candidate versions is attempted, conflicts among
the candidate versions may be detected, and a cooperative session among
the developers of the candidate versions will have to be initiated. The purpose of the cooperative session is to negotiate a new version of the development object.

The negotiations are based on the conflicting candidates. The session may be synchronous or asynchronous. Tool support is provided through multi-user editing capabilities.

In the synchronous case all candidate versions are available to the involved parties during the session through a multi window interface. A developer may comment all proposals. Various candidates may be compared, both on a non-executable (textual or diagrammatic) and an executable (by simulation or execution) basis. Cut and paste among windows to achieve a reconciled version is supported.

In the asynchronous case one have to rely on written communication in the form of comments added, and cut and paste versions, to convey opinions among the developers involved in the negotiation process.

A successful negotiation session results in an agreement on one reconciled version that will be the next version of the development object in question. This version will be stored in the specification database, and the development transaction is terminated.

Suppose that development transaction Trans-1 is ready to check-in the development object at "Date = 13.01.92 : 15:05:50". Additionally we assume that the developer responsible for transaction Trans-1 is also responsible for Replenish Stock. This means that in the end it is the owner of Trans-1 who determines the time of check-in. When Trans-1 issues the request for check-in, the check-out contract for the development object is inspected. All transactions that is registered as having checked out the development object is notified of the request for check-in. Suppose that transaction Trans-2 requests a one week postponement before merging, the owner of Trans-1, being the developer responsible for the development object, is at liberty to grant or reject the request. When merging potentially is needed, i.e. when several candidates for the next version of the development object exists, there are two possible outcomes:

- The developer responsible for the object picks one of the candidates. The chosen candidate will be checked back into the repository as the next version of the development object.

- The developer responsible decides to merge the candidates. There are two ways of attempting to merge the candidates:
  - Try an automatic merging of the candidates. If overlap is detected, initiate a cooperative session to negotiate a new version.
  - Initiate a cooperative session to negotiate a new version directly without trying to merge automatically first.

When the merging process is finished, the result is checked back into the specification database as Replenish Stock 1.3, i.e. filling the placeholder for this version of the development object.
5.3. Development transactions

A development transaction is the task of developing a new version of a development object. A development transaction is initiated, controlled and terminated by a developer. It takes as input a version of a development object, and involves the use of several automated tools by intelligent human agents, to produce an output in the form of a new version of the development object. This version will update the specification repository. The development transactions closely reflect the work task structure of the development project. Development transactions may therefore be of considerable length.

5.3.1. Properties of development transactions

Development transactions will differ from database transactions in several ways [98].

- The controlling agent for a development transaction is human.
- Strict serialisability is not achievable.
- Rollback is normally not feasible.
- Update conflicts cannot generally be handled automatically.

The development transaction provides a framework to perform consistent updates of the specification repository by several developers [92],[30]. In determining a working practise for the development transaction we firstly touch briefly upon the concept of long and nested transactions before we discuss the check-out/check-in operations.

Long transactions

Development transactions are examples of transactions where elapsed time from check-out till check-in may be in the range of days and weeks. E.g., the task of performing a modification in a development object may take a couple of weeks. During this time the development object in question is unstable. It is in the process of being replaced by a new version. When a development object is in an unstable state, access to it by other developers is normally prohibited. However, the potentially extensive duration of the transactions indicate that one must provide adequate facilities for sharing of specifications during transactions. In the case of overlap between specifications worked on by different developers, the strategy of locking utilised by standard database systems is clearly too restrictive and thus not feasible.

Nested transactions

Development transactions will reflect the work task structure of the development project. Most development projects will be organised into a hierarchy of tasks with associated milestones. It is therefore necessary to support nested transactions, i.e. transactions and sub transactions, to properly reflect the hierarchical organisation of development projects.
A sub transaction is constrained relative to its superior transaction both in elapsed time and in scope. All sub transactions must terminate before their superior may terminate. However, a sub transaction may be wider in scope than its superior provided that the new objects checked out aren't going to be modified. The scope is in terms of the specifications checked out by the transaction.

A transaction cannot terminate before all sub transactions have terminated. If the transaction terminates before its sub transactions terminate, one will in general not have met the conditions on the development objects checked out to the transaction that were agreed upon when the transaction was initiated. In that case the conditions that have not been met will have to be modified and the responsibilities divided among and turned over to the sub transactions. This is in fact equivalent to terminating the old transaction together with its sub transactions, and the creation of a new set of transactions with a new set of conditions to fulfil the original commitments of the original transaction. If this does not happen, the transaction would have failed. Thus, it makes sense to say that a sub transaction cannot live longer than its superior.

A sub transaction will not be allowed to operate on, i.e. modify, any development objects that were outside the scope of the superior transaction. However, objects other than those checked out by the superior may be checked out to the sub transaction provided that they are not going to be modified. If the sub transactions were allowed to modify objects that were not within the scope of the superior transaction this would be inconsistent with the conditions agreed upon when the superior was initiated. This is clearly undesirable. Thus, it makes sense to say that a sub transaction may only modify objects that may be modified by its superior.

5.3.2. Check-out and check-in

The development transaction is implemented by the check-out and the check-in operations. Check-out creates a private repository (or workspace) of the development objects that the transaction is going to operate on. Check-in returns the development object in the private repository to the shared repository. Modifications made to a development object when it is checked out are considered private and kept separate until check-in time.

There are several issues related to the actual semantics of the check-out and check-in operations that have to be decided on. We proceed to discuss the following: Authorisation on checked out development objects, relaxed authorisation rights, propagation of authorisation rights in the development object hierarchy, single or multi repository check-out, and individual or total check-in of development objects.

Authorisation on checked out development objects

The most frequently used authorisation rights are the shared and exclusive lock modes used in databases. This is obviously sufficient for the average database transaction, but for long lived development transactions the solution is too restrictive. The following four authorisation rights, may prove more appropriate [142]: Read only, access, modify, delete. The au-
torisation right characterise the relationship between a development object and a developer.

- Authorisation *read only* implies that development objects can be read, but not changed or referenced. Associations to the object cannot be added or deleted.

- Authorisation *access* implies that development objects cannot be deleted or changed. Associations with the object can be added and deleted.

- Authorisation *modify* implies that objects can be changed but not deleted. Associations with the object can be added and deleted if the deletion does not imply deletion of the object.

- Authorisation *delete* implies that development objects can be deleted. If the object is used by other objects it may be necessary to delete references to the object before deleting it.

Figure 5-25 depicts the compatibility matrix for these authorisation rights. E.g., if a developer has been granted *delete* authorisation to a development object, then other developers may have at most *read only* authorisation to this development object.

<table>
<thead>
<tr>
<th>Max. authorisation granted</th>
<th>Highest authorisation other developers may have</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete</td>
<td>Read only</td>
</tr>
<tr>
<td>Modify</td>
<td>Access</td>
</tr>
<tr>
<td>Access</td>
<td>Modify</td>
</tr>
<tr>
<td>Read only</td>
<td>Delete</td>
</tr>
</tbody>
</table>

Fig. 5-25: Authorisation rights compatibility matrix

**Relaxed authorisation rights**

We propose to relax the authorisation rights such that several developers may check out a development object with the intention of modifying it, i.e. check-out with authorisation *modify or delete*. At check-in time potential candidates for the next version of the development object, candidate versions, will be checked for overlap. Where overlap is detected a cooperative session among the involved developers is initiated. The purpose of the cooperative session is to negotiate a new version of the development object. The negotiations are based on the conflicting candidate versions. The negotiation session may be synchronous or asynchronous [44].

**Propagation of authorisation rights in the development object hierarchy**

Authorisation rights will be automatically propagated through the development object hierarchy. It is possible to override the propagated authorisation rights in the following way: The developer may be granted the same or more restrictive authorisation rights to subordinate development ob-
jects. I.e., if a developer has delete authorisation to a development object, he may have either delete, modify, access, or read only authorisation to its subordinates. To grant a developer less restrictive authorisation rights to subordinate development objects, one must make sure that conflicts with other developers accessing the same development object does not arise. I.e., if a developer has read only authorisation to a development object, he can only be granted access, modify, or delete authorisation to its subordinates provided that this is not in conflict with authorisation rights being held by other developers on the same development objects. To grant a developer an authorisation right that is in conflict with the compatibility matrix in figure 5-25, one must first change the authorisation rights of other developers.

Single or multi repository check-out

Multi repository check-out means that one is allowed to check-out development objects from several shared repositories into a private sub-repository. Single repository check-out means that one is obliged to have a one-to-one relationship between a sub-repository and a shared repository.

The single repository check-out solution is the simpler. There is no need to keep track of where different development objects in the private repository belong, because they all belong to the same repository. However, the solution limits the possibility to share development objects among several repositories. It seems artificial to introduce this limitation. The added cost of allowing multi repository check-out is to keep information on where the development objects belong. This is a reasonable price for added flexibility.

Individual or total check-in of development objects

Individual check-in of objects means that the development objects may be checked back into the shared repository on individual basis. Total check-in of objects means that development objects that were checked-out together at the beginning of the transaction, are checked back into the repository together. This usually happens as an integral part of the termination of the development transaction.

A transaction is an assembly of work that logically belong together. The transaction is started at one point in time, and it should be terminated at one point in time. At that time all development objects manipulated by the transaction should be checked back into the shared repository.

5.4. Development teams

Development of complex information systems necessarily involves several systems developers working in a team. The development team may be geographically distributed. The team collaborates in the process of developing a new version of a development object. Team work facilitates the undertaking of large tasks, at the cost of increased communication and coordination overhead.

In developing a new version of a development object several developers may be involved, submitting different contributions which need to be considered when assembling the new version. We propose a scheme that will
allow the individual developers involved in the process of merging the different contributions to browse each others proposals. A developer may thus view the work of other developers while working on their own contribution [9].

To support this team process we need to take administrative measures to avoid chaos when the version is checked back in. A consequence of allowing several developers to work in parallel based on the same development object version, is the need to merge the different contributions made by the developers. The result of merging different contributions constitutes the next version of the development object. If one restricts the number of concurrent developers to one, there is no need for merging.

5.4.1. Synchronising transactions

Although a system is described as a hierarchy of system components, dependencies between components that are not directly related in the hierarchy will exist. During a development transaction there may be a need to synchronise, or coordinate, the work taking place within a transaction, with on-going development work on development objects checked out to other transactions. Work taking place in one transaction may depend on the results of other transactions. Synchronising transactions may be needed both when there is overlap among the development objects checked out to the different transactions and when no overlap exists.

The synchronisation of transactions is done through the use of check-out contracts and transaction logs.

- The transaction log is a record of the development state of a transaction and its related development objects. A transaction log is associated with a transaction. It contains information on the development objects currently checked out by the transaction, dependencies on other transactions, in addition to a record of the development history within the transaction. The transaction log is established at check-out time.

- The check-out contract is a record of the development state of a development object. A check-out contract is associated with a version of a development object. It contains information on responsible developer, development status, dependencies on other objects, and the development transactions that currently has checked out the object. The check-out contract is established when the placeholder for the version of the development object is created.

Dependencies between a specific transaction and other transactions that currently have checked out development objects that overlap or are dependent on development objects currently checked out by the specific transaction will be recorded in the transaction log. When checking out a development object the check-out contract is inspected and the transaction log is updated together with notification of the developer, of any other developers having checked out the same object and any dependencies between this object and any other objects. The check-out contract is updated with information on dependencies, if any, between this object and any other objects currently checked out. This information is supplied by the developer responsible for the transaction.
At check-in time the transaction is checked for consistency against the existing contract and contracts on related transactions. Provisions are made for merging different proposals.

We introduce the possibility for transactions to publish local revisions of the development objects it has checked out, and for transactions to subscribe to local revisions of development objects of other transactions. A subscription is negotiated and established as part of updating the transaction log and the check-out contract when checking out an object. When a transaction publish a local revision of a development object, any transaction having subscribed to this development object will automatically be informed of, and have access to, the new release of an intermediary version.

**Merging of candidate versions**

At check-in time a development transaction will possibly check several development objects back into the specification repository with the purpose of updating the repository with new versions of the development objects. In the general case there will be several candidates provided by several transactions, for the new version of these development objects. A strategy for merging these candidate versions will be necessary to be able to support teams in systems development. The goodness of the merging strategy is heavily dependent on the conflict resolution mechanism applied when overlap between candidates have been detected.

There are several policies for handling the situation where several candidate versions of a development object exists:

- **Priority** Let the first, last, or some other candidate with priority win. This may waste weeks of work, and is clearly not acceptable.

- **Rollback** Where conflicting overlap exists, roll back the transaction. This may waste weeks of work, and is clearly not acceptable.

- **Locking** Going down one level in granularity together with an adequate set of authorisation rights, may remove the problem.

- **Merging** Generalised merging of several candidate versions. May be semi automatic or entirely manual.

The only reasonable solution is to provide merging of candidate versions. Some observations on merging of candidate versions:

- A human developer exists as the intelligent agent behind the development transaction.

- Coarse grained version control may lead to flagging of update conflicts which by deeper semantic analysis are found to be non-existent.

The support of this merging process is very much an administrative problem. The bottom line conflict resolution mechanism when merging, is detection and notification of potential conflict, and request for manual intervention. As deeper knowledge of the models involved is gained, and experimentation with alternative solutions can be carried out, a better and more informed strategy for merging can be provided.

A strategy for supporting the merging of candidate versions, is as follows: When a transaction is initiated and a set of development objects are checked out for further development, placeholders for new versions of the
development objects are created. The *placeholders* serve the purpose of defining whether the new version is revision related or variant related to the version that has been checked out. A transaction log is established for this transaction, and a check-out contract for the object (the placeholder) is created. The check-out contract will require that the developer points out dependencies between development objects that have been checked out by this transaction, and development objects that have been checked out by other transactions. The dependencies may exist both when the development objects overlap and when they do not overlap. During a transaction the developer may create a chain of revisions of the development objects that have been checked out. When the chain of development revisions necessary to do the modification is completed, the developer may terminate the transaction by checking the new version of the development objects back into the shared repository, thus attempting to fill the placeholder. If transactions that have checked out an overlapping set of development objects exist, we potentially have a conflict. A potential conflict triggers the conflict resolution mechanism.

The chains of intermediary revisions created during the development transaction can be handled as intermediary structures on the version graphs [8]. The intermediary structures may be discarded after successful check-in of the new version. The check-out contract associated with the placeholder may also be discarded. However, if it is desirable to keep the full development history then the intermediary structures and the check-out contract must be stored.

![Fig. 5-26: Multi transaction check-out/check-in in a version graph](image)

Figure 5-26 depicts a version graph where *S 1.1* have been checked out by three transactions. Each transaction have developed intermediary revisions of the development object local to the transaction. The checked out development object is identified by the name *S 1.1(τ, λ)*, where *τ* indicates that it belongs to transaction *τ*, and *λ* is the revision number local to the transaction. When either of the transactions attempt to check-in the devel-
opment object, merging is attempted. The placeholder for the new version of \( S, S^1,2 \), is grey in the graphical representation in figure 5-26. When the placeholder is filled, it will be changed to white.

5.4.2. Conflict resolution

When several candidates for a new version of a development object exist, merging of the candidates must be performed [9]. All developers having checked out the said version are notified, and a suitable time for attempted merging is negotiated. The time to merge is determined by the developer responsible for the development object. Responsibility for development objects is determined by the allocation of manpower in the project organisation. Some developer may need an extra day or an extra week before he is ready to merge with the others. This will have to be negotiated among all developers involved, and sanctioned by the developer responsible for the development object. The responsible developer will still have priority in determining the time of merging, so that the work will not be unnecessarily delayed.

Merging of the candidate versions is attempted. When a conflict has been detected a cooperative session among the developers of the candidate versions is initiated. The purpose of the cooperative session is to negotiate a new version of the development object. I.e. the goal of the negotiations is to achieve consensus on a new version of the development object. The negotiations are based on the conflicting candidates. The negotiation session may be synchronous or asynchronous [44]. Tool support may be provided through the integration of multi user editing capabilities as described in for instance ForComment [116], Collaborative Editing System [61], Shared Book [97], or Quilt [50][96]. Added functionality to the negotiation support may be obtained by integrating synchronous video conferencing [77] and multi media annotation [52].

Synchronous negotiations

In the synchronous case all candidate versions are available to all involved developers during the negotiation session. The candidate versions are available through a shared workspace for comments and comparisons. A developer may comment on any proposal. Comparisons may be based on the textual or diagrammatic specifications, i.e. a non-executable basis, as well as by simulation or execution of “executable” components. The synchronous shared workspace supports several modes of working [77]:

- The public screen and public desktop allow all developers involved in the negotiation to view one single developer's screen or desktop.

This mode implements multicasting of one developer's screen or desktop to all the other members of the negotiation. While the screen or desktop may only be manipulated by its owner this mode may be used when a developer is explaining a diagram or arguing a point to the other developers involved in the negotiation. To support the spoken explanation or arguments the developer may simultaneously be drawing sketches or extending the diagrams on the desktop or screen and pointing to objects in a sketch or diagram.
• *Desktop on screen* and *desktop on desktop* allow developers to draw sketches and comment the contents of other developer's screens or desktops.

This mode implements translucent overlay of live video images of individual screens or desktops. It offers an efficient possibility of interactively commenting the contents of another developer's screen or desktop. Images of desktops may overlay screens or images of other desktops. A screen or desktop image that is overlaid by an image of a desktop may already have images of other desktops overlaid. Experimental evidence indicates that users easily differentiate up to three overlays [77]. When there are more than three overlays differentiating between them may be alleviated by introducing colour coding of the different layers.

• The *shared tool* mode allows all developers involved in the negotiation to simultaneously view and manipulate (edit) one or several development object candidates using the appropriate tool on their private workstations.

This mode offers joint editing functionality on a development object. A telepointer is available for global pointing. There is no special floor control policy. One relies on the informal social protocol agreed by the developers through synchronous voice or video communication. The tool to be used is dependent on what tool was used to develop the development object candidates that are the subject of the negotiation process.

The usefulness of these modes are dependent on the presence of two additional factors; the addition of synchronised sound or video and the possibility of flexible switching between modes.

Firstly, all three modes must be used together with synchronised sound, e.g., using speakerphones. It would be impossible to e.g conduct interactive joint editing without possibilities of synchronising actions through speech. Better still would be to use synchronised video to capture facial cast and body language as well as speech. This will enable synchronised negotiation sessions to take place in geographically distributed environments. It is possible for the developers to stay in their respective offices and still have a meeting with the feeling of “eye to eye” contact. In addition to the obvious benefits of not having to move from your office where you have your work context, e.g., your papers, files, workstation etc., distributed meetings offer the added flexibility of allowing the participants to enter and leave the meeting at different times during the meeting without too much disturbance to the rest of the participants. The alternative to distributed meetings is either to have the participants meet in one room or conduct an asynchronous negotiation session.

Secondly, it is necessary that switching between modes is flexible. For example, a developer may operate in *public screen* mode to explain some issues in his development object candidate. When somebody wants to comment on the object currently in the public screen, one may switch to *desktop on screen* mode to be able to make a sketch and do some pointing to explain the comment. Other developers may chain on to the comments so far to give their comments. This may be done by simply overlaying their own desktops on top of the ones that are already present. In the end an agreed position will be reached, and a consolidated proposal will be devel-
oped. The consolidated proposal may typically be in the form of sketches and comments overlaid on somebody’s screen or desktop. It is then possible to enter into the shared tool mode to jointly perform the necessary modifications to the original development object candidate as indicated by the sketches in the overlaid desktops.

There are some projected negative aspects of the proposed synchronous modes of operation:

- The public screen and public desktop mode offers a plain “one way” communication with much the same features as traditional lectures. The only possible feedback is through spoken comments. However, this may be improved when necessary by switching to an overlay mode where additional input may be given by those who wish to add comments.

- The desktop on screen or desktop on desktop mode offers indirect drawing an pointing on an overlaid desktop by hand. This needs time and effort to get used to. This is similar to the learning process required for learning to use indirect pointing devices, e.g., a mouse.

- The desktop on screen or desktop on desktop mode offers the use of many overlays. This may cause confusion. This may be remedied by using colour coding of layers, dimming of layers, or conceptually removing layers from the picture.

- The desktop on screen or desktop on desktop mode offers the results of the negotiations as a many layered object. The consensus is not directly recorded in the “primary object”. Individual contributions are overlaid to make the complete picture and they are physically stored in different places and on different media. One will therefore have to perform the modifications that were agreed upon on the primary object in a separate editing session.

- The shared tool mode implements an informal social floor control. This may cause confusion when several developers input data at the same time. We wish to have a flexible shared tool mode, consequently we do not wish to implement a more rigorous floor control policy which restricts concurrent input, e.g., a token based approach.

Asynchronous negotiations

In the asynchronous case one has to rely on written communication in the form of comments and cut-and-paste versions, and multi media annotations to the candidate development objects to convey opinions among the developers that are involved in the negotiation process.

The multi media messages passed between the developers are based on the candidate versions. The messages will contain annotations to one or several candidate versions in the form of synchronised pointing, drawing, writing, and speaking [52]. To annotate a local version of a development object, e.g., in the form of a Data Flow diagram, one would add a transparent annotation layer to the object. The annotation layer allows the developer to point, write, draw sketches, and add voice comments to the diagram by using a voice handset, a tablet, and a “pencil” stylus. The key- board may be used for lengthy input of text.
The multimedia annotations may be played back in synchronous mode. This will play back the voice comments, the stylus cursor’s movement, as well as the hand drawn and typed messages. For instance, when reviewing a Data Flow diagram one may point to a specific process and say, “This process consumes data from our Order History files. This seems to have been omitted in this candidate version.”

People are accustomed to annotating paper documents. The combination of written and spoken annotation, commonly used only in meetings, will increase the information content and precision in asynchronous communication. Comments may be added as the multi media annotated development object pass from one developer to the next. The object holds the accumulated signatures and comments of several developers involved in the negotiation on the candidate objects. One will observe a conversation unfolding over time as the objects are passed back and forth among developers for continued annotations.

There are some projected negative aspects of the proposed asynchronous annotation scheme:

- The result of the asynchronous annotation session is a many layered multi media document. Navigating among a high number of layers may be confusing. The navigation and playback of the recorded annotations may be alleviated by disciplined, focused, and too the point annotations.

- The result of the asynchronous annotation session is an annotated multi media document. The modifications proposed are not implemented in the primary object. The modifications that have been agreed upon must therefore be implemented in a separate editing session. This may be done either as a synchronous group task through a joint editing session, or by the developer responsible for the primary object.

![Many layered object during negotiations](image)

**Fig. 5-27: Many layered object during negotiations**

**Many layered objects**

During a negotiation session, synchronous or asynchronous, the development object that is the “object of” the negotiation will be a many-layered object, as depicted in figure 5-27. The primary layer will consist of the local development object itself. There will be a number of transparent secondary layers containing overlaid desktops or screens and multi media annotations. The different overlays may have been added in synchronous
or asynchronous mode. The developers may dim or conceptually remove any layer to make it easier to interpret the possibly complex picture that may arise from the existence of many layers. Colour coding of different layers will also improve the readability of the many layered object.

A successful negotiation session results in an agreement on the next version of the development object. The reconciled version will be the next version of the development object and update the repository. This concludes the merging operation, and the development transactions that are involved may terminate.

5.5. Chapter summary

This chapter has presented a framework for management of change of information system models. The framework comprises a version and configuration management scheme which is able to model information system models of hierarchic and flat types. The framework includes the solved by construct which allow for inconsistencies in the information system models. A scheme which guarantees global name uniqueness has been presented. The framework further includes the concept of development transactions which allows asynchronous modes of working and coordination among the members of the development team. A relaxed set of authorisation rights to development objects provides true sharing of objects, and the publish and subscribe facilities helps monitor potential conflicts among the transactions. Finally, the framework includes facilities for supporting synchronisation of work performed by the individual members of the development team by synchronisation of development transactions. Negotiation among development transactions may be synchronous or asynchronous and is supported by multimedia cooperation techniques such as mixing of live video from multiple sources, annotation, and the concept of many layered development objects.

This provides true sharing of information system models among development team members, monitoring of conflicts among the development transactions, and support for conflict resolution in either synchronous or asynchronous modes.
6. Change support for complex models

The specifications of non-trivial systems are complex and contain large numbers of detail. The number of specification details increases rapidly as we approach the implementation level. As the amount of detailed specifications increases, the possibility decreases for humans to understand their implications. The specification may become too large to be grasped in a short time. Neither systems developers nor end users can perceive large numbers of detailed specifications unless the specifications are considerably simplified, for example by hiding details. The specification may comprise detail that is of little interest to the actors involved in some of the deliberations that take place when modifications are specified. Irrelevant details must consequently be removed so that the actors' minds can be more easily focused on relevant issues during the modification process [140].

This chapter develops such modifications and additions to our framework for cooperative work management that are forced by these requirements to the modification process that are relevant when dealing with complex models. Special attention is given to the problem of support for change when details are abstracted away from the specifications. Specifically, the implications on the version and configuration management scheme and the development transaction scheme are considered. A conceptual model of the cooperative work management framework is presented.

6.1. Abstractions

An abstraction of a development object is a view of the object where details have been removed from the specification. In this way the remaining part of the specification, i.e. that which have not been removed, is highlighted.

The concept of abstraction is defined in Collins Cobuild Essential English Dictionary [26] as follows:

"An abstraction is a general idea rather than one relating to a particular object, person, or situation."

Harrap's Easy English Dictionary [56] gives the following definition:

"(a) removing; stealing, (b) vague idea"

Based on these definitions, we define an abstraction of a development object, in systems development terms:

"An abstraction of a development object is a view of the object where one seeks to convey the general ideas of the development object through removal of details."

There are many possible ways of removing details from specifications, hence many different abstractions can be derived from a development object. The process of abstraction may range from an entirely manual process to an automatic transformation. In the manual process the systems developer would explicitly mark every component that is supposed to be abstracted away or every combination of components that is supposed to be combined into a compound object. For example, combining simple data flows in a data flow diagram into compound flows would have to be done
manually by marking each flow that is to be included in the compound flow. Some types of abstractions can be performed by deterministic automatic transformations that just needs triggering by the systems developer. For example, removing all attributes in an Entity Relationship diagram is a simple deterministic operation that easily could be triggered by choosing “Remove attributes” from a menu of possible categories or types of abstractions.

It is desirable to support systems development, including modification of systems, by providing the systems developers with abstractions so that a simplified picture of the system can be used for communication purposes. Significant parts of the modification may then be made on abstracted specifications. When the modification process has reached a stage where one is about to add details to the abstracted development object that go beyond what the abstraction will support, e.g., adding attributes to an entity in an Entity Relationship diagram where the attributes have been abstracted away, it is desirable to continue the development on development objects with fully detailed specifications. For example, if you work on an abstraction of a Data Flow diagram where the data stores have been removed, and you want to add a data flow from a process to a data store, obviously the abstraction will not support nor permit this. Thus, adding this data flow will have to take place on a Data Flow diagram with data stores. In this example, to continue the work on the Data Flow diagram one has to return from working on the abstraction to working on the data flow diagram with all the details.

![Diagram](image1)

**Fig. 6-1: Forming compound data flows**

<table>
<thead>
<tr>
<th>type(CF-1):</th>
<th>Compound Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>components(CF-1):</td>
<td>f1, f2</td>
</tr>
<tr>
<td>connections(CF-1):</td>
<td>P1.f1 $\rightarrow$ CF-1, P2.f2 $\rightarrow$ CF-1, CF-1 $\rightarrow$ P3</td>
</tr>
</tbody>
</table>

![Diagram](image2)

**Fig. 6-2: Definition of compound flow, CF1**

When the modification process has reached a level of detail that goes beyond what the abstraction will support, the development will have to continue on objects with all the details. This means that the abstracted develop-
velopment object will have to be brought back to the same level of detail as before the abstraction was applied. After doing this, the development object will contain components that have not been changed, components that have been changed, deleted components, and new components. For the components that was never changed, the details that was abstracted away will be directly available from the version of the development object that existed before the abstraction was applied. The changed and new components will have to be further augmented to be brought to the same level of detail as the components that was never changed. The components that were deleted in the abstracted object, will have to be removed also in the version of the object with all the details. This requires that abstractions as well as the fully detailed specifications, have to be supported by e.g., version control.

We view abstractions as temporary development objects serving two purposes: On one hand they are created as a means of enhancing the communication between end users and analysts, and on the other hand they act as a means of providing the analysts themselves with an improved overview of the system through specifications less disturbed with details. They will be used quite intensively during their period of existence. Thus, storing them in full will be worth the extra storage costs.

Figure 6-1 depicts a simple abstraction which consists of composing the data flows f1 and f2 into a compound data flow called CF-1.

A compound flow is defined as the composition of several data flows into a compound flow. It may be defined as indicated in figure 6-2.

The use of abstractions, and more generally the use of simplification techniques in information system development is discussed in [126] and further elaborated in a forthcoming thesis [127].

Why abstractions?

Contemporary systems development models must fulfil two goals: Firstly, the specifications must be able to act as an efficient and effective communications medium between the different parties in a systems development project. I.e. the specifications created must be properly understood by the systems analysts, the designers, the programmers, and the end-users, and at the same time sufficiently expressive to allow non-ambiguous expression of requirements and proposed solutions by the different parties in the project. Secondly, the specifications must be able to act as the basis for the generation or development of efficient and effective executable code to implement the requirements of the user community.

The systems development methods that facilitate executable specifications, rapid prototyping, or automatic generation of executable code are based on developing rich specifications. The specifications are developed in an iterating manner. Specifications are successively augmented by details on each development step, until the specifications contain sufficient detail to facilitate execution or generation of executable code. It readily follows that details will abound.

In addition to being the basis for execution and generation of executable code, the specifications must also support the communication between the different parties in the systems development project. The kind of rich
specifications that are discussed in this text are not helpful in this communication process. The number of details occurring in the specifications is very high. Together with an extensive degree of formality in terms of mathematical logic, this constitutes major prohibitive factors for efficient and effective communication. Abstractions of the specifications may be helpful.

Suitable mechanisms for abstracting away details of various kinds will greatly aid the systems developers in properly understanding and manipulating the specifications during the modification process. Abstractions further make it possible to postpone the specification of details during the development process. Thus, one is allowed to concentrate on introducing details in the specifications on different levels of abstraction at different stages during development, thus simplifying the development process.

The example revisited

As an example, consider Replenish Stock version 1.2, that is depicted in figure 5-4. Figure 6-3 depicts an abstraction of Replenish Stock 1.2 where data flows have been composed into compound flows.

The fact that this is an abstraction is indicated by the \(<\alpha>\) added to the name of the development object. The type of the abstraction is indicated by giving it an abstraction identifier, \(\alpha\), in this case the number 1. Hence, this specific abstraction, characterised by forming compound flows, is identified by adding \(<I>\) to the development object name.

![Diagram of Replenish Stock 1.2](image)

Fig. 6-3: Replenish Stock 1.2<1>

To avoid unnecessary cluttering of the diagram, the names of the constituents of a compound flow are not listed in the abstracted diagram. The compound flows are labelled \(CF-\phi\) (Compound Flow \(\phi\)), where \(\phi\) is unique among the compound flows appearing in the diagram. The constituents of a compound flow can be obtained by double clicking on the flow, thus opening a window containing information on which flows the compound flow contain. The compound flows \(CF-1\) and \(CF-2\) in figure 6-3 may be defined as indicated in figures 6-4 and 6-5.
The processes P1, Estimate Future Need, and P2, Find Best Supplier were decomposed in figures 5-11 and 5-12. Abstractions of P1 and P2 where data flows have been composed into compound flows are depicted in figures 6-6 and 6-9. The development objects are given the names Estimate Future Need 1.0<1> and Find Best Supplier 1.1<1>.

**Fig. 6-4:** Definition of compound flow, CF-1

**Fig. 6-5:** Definition of compound flow, CF-2

**Fig. 6-6:** Estimate Future Need 1.0<1>
the external entity *Stock Clerk* to process P1 in figure 6-3, is referenced by *P1.Replenish Part* within the decomposition of process P1.

*CF-1* in figure 6-6, is a combination of the two flows *Order History* and *Replenish Part*, terminating at P1.2. *CF-2* in figure 6-6, is a combination of two flows flowing from *P1.1 to P1.2* and *P1.3* respectively.

<table>
<thead>
<tr>
<th><strong>type(CF-1):</strong></th>
<th>Compound Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>components(CF-1):</strong></td>
<td>Replenish Part, Order History</td>
</tr>
<tr>
<td><strong>connections(CF-2):</strong></td>
<td>P1.Replenish Part → CF-1, P1.Order History → CF-1, CF-1 → P1.2</td>
</tr>
</tbody>
</table>

**Fig. 6-7: Definition of compound flow, CF-1**

<table>
<thead>
<tr>
<th><strong>type(CF-2):</strong></th>
<th>Compound Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>components(CF-2):</strong></td>
<td>Orders Pending(i), Orders Pending(j)</td>
</tr>
</tbody>
</table>

**Fig. 6-8: Definition of compound flow, CF-2**

**Fig. 6-9: Find Best Supplier 1.1<1>**

*CF-1* and *CF-2* in figure 6-9 are defined in figures 6-10 and 6-11. *CF-1* in figure 6-9, is a combination of two parallel flows, *Order Quantity* and *Able to Deliver*, flowing from P2.2 to P2.3. *CF-2* in figure 6-9, is a combina-
tion of two parallel flows, *Order Quantity* and *Chosen Supplier*, flowing from P2.2 to P3.

**Fig. 6-10: Definition of compound flow, CF-1**

| type(CF-1): | Compound Flow |
| components(CF-1): | Order Quantity, Able to Deliver |
| connections(CF-1): | P2.2.Order Quantity → CF-1, |
| | P2.2.Able to Deliver → CF-1, |
| | CF-1 → P2.3 |

**Fig. 6-11: Definition of compound flow, CF-2**

| type(CF-2): | Compound Flow |
| components(CF-2): | Order Quantity, Chosen Supplier |
| connections(CF-2): | P2.3.Order Quantity → CF-2, |
| | P2.3.Chosen Supplier → CF-2, |
| | CF-2 → P2 |

When comparing figure 6-3 with figures 6-6 and 6-9, there are two remarks to be made:

Firstly, note that the compound input flow into process P1 in figure 6-3, does not correspond to the compound flow, *CF-1*, used inside the decomposition of process P1 in figure 6-6. Note also that the compound flow flowing out of process P2 in figure 6-3, corresponds to the compound flow, *CF-2*, inside the decomposition of process P2 in figure 6-9. This means that when a process is decomposed, a compound flow flowing into or out of the process, may or may not correspond to compound flows being defined and used within the decomposed process. I.e. a compound input flow may be directly used as a compound flow within the decomposition of the process, or the compound flow may be decomposed on the process border and its constituents used to form new compound flow combinations within the decomposed process. Similarly, a compound output flow may correspond one-to-one with a compound flow within the decomposition of the process, or may be compositions of different flows, compound or not, from within the decomposition.

The consequence of the above is that for the flow abstractions to function there is a need to develop a kind of socket system for compound flows. The socket is the connection point for compound flows on the process border. This facilitates decomposition of compound flows, with subsequent recombination and composition of new compound flows when a process is decomposed. The sockets are indicated by the filled rectangles occurring on the starting points and termination points of the compound flows in the figures 6-3, 6-6, and 6-9.

Secondly, note that compound flows constitutes a simplification of the flow structure in terms of collecting simple flows into compound flows. The
constituents of a compound flow may originate from different diagram components (processes, data stores, and external agents), and terminate at different diagram components. For example, the compound flow \textit{CF-2} in figure 6-6, originates from process P1.1, and has constituents that terminate at process P1.2 and process P1.3 respectively. On the other hand, compound flow \textit{CF-1} in figure 6-9, originates from one single process, P2.2, and terminates at one single process, P2.3.

The consequence of the above is that for the flow abstractions to function there is a need to develop a pipe system that facilitate merging and splitting of data flows. The pipe system, i.e. the compound flow, would collect simple and compound flows from different sources into one compound flow. The pipe system would also allow simple and compound flows to be split from the compound flow and terminate at different sources. The pipe system (compound flows) are indicated by bold arrows in figures 6-3, 6-6, and 6-9.

6.2. Properties of abstractions

Our definition of abstractions means that there are many types of abstractions highlighting or suppressing different features of the models. Furthermore there will be several different ways of forming an abstraction of a specific type. Finally, for hierarchical models some abstraction types will have a scope of effect which covers more than one level in the hierarchy.

Several possible abstractions

An abstraction of a development object is a view of the object where details have been removed from the specification. There are many possible ways of removing details from the specifications, hence many different abstractions can be derived from a development object. It is therefore possible for a version of a development object to have several abstractions related to it.

The process of abstraction may range from an entirely manual process to an automatic transformation. In the manual process the systems developer would explicitly mark every component that is supposed to be abstracted away or every combination of components that is supposed to be combined into a compound object. For example, combining simple data flows in a data flow diagram into compound flows would have to be done manually by marking each flow that is to be included in the compound flow. Some types of abstractions can be performed by deterministic automatic transformations that just needs triggering by the systems developer. For example, removing all attributes in an Entity Relationship diagram is a simple deterministic operation that easily could be triggered by choosing "Remove attributes" from a menu of possible categories or types of abstractions.

As an example, consider the Data Flow models. Examples of abstractions, i.e. ways of reducing complexity, that can be performed on a Data Flow model, are:

- Composing flows into compound flows
- Hiding of data stores
- Hiding of external entities
- Hiding of data flows

The first of these abstractions needs to be done manually, while the three latter may be done automatically. The transformations needed to do the three latter abstractions may be triggered by the developer from a menu of possible categories of abstractions. The menu that allows the developer to specify the details of the abstraction may be designed as a multiple choice menu. In the menu one or more boxes may be ticked, thus indicating whether the abstraction feature is included or not. Figure 6-12 depicts an example of a menu where the current choice is *Hiding of data stores* and *Hiding of data flows*.

![Forming Abstractions](image)

Fig. 6-12: Multiple choice menu to form abstractions

It is possible to define and identify several alternative abstractions related to one version of a development object. Figure 6-13 depicts a situation where *Replenish Stock 1.2* have three different abstractions of different types associated. They are called *RS 1.2<1>.0*, *RS 1.2<2>.0*, and *RS 1.2<3>.0* for short.

![Diagram](image)

Fig. 6-13: Three different abstractions of one development object version

The different abstractions are formed using different transformations. The abstractions are different, and serve different needs. Imagine for example, that *RS 1.2<1>.0* was formed by composing flows into compound flows, as was depicted in figure 6-3. This abstraction puts emphasis on simplifying the flow structure, i.e. the interrelationship between processes, and highlights the high level data flow in the model. Secondly,
RS 1.2<2>.0 may have been formed by choosing *Hiding of data stores*. This abstraction puts emphasis on the process-flow part of the model. The abstraction is depicted in figure 6-14.

![Diagram of Replenish Stock 1.2<2>.0](image)

Fig. 6-14: *Replenish Stock 1.2<2>.0*

Finally, RS 1.2<3>.0 may have been formed by choosing *Hiding of data stores* and *Hiding of data flows* as was the case in the menu depicted in figure 6-12. This abstraction highlights the “actors” of the information system being modelled, in terms of external entities and processes, as independent entities with known (processes) or unknown (external entities) data processing tasks. The abstraction is depicted in figure 6-15.

![Diagram of Replenish Stock 1.2<3>.0](image)

Fig. 6-15: *Replenish Stock 1.2<3>.0*

**Forming abstractions is a user determined process**

Some types of abstractions are such that the process of arriving at the abstraction is determined by the user. This means that the transformations necessary to obtain the desired abstraction are not straightforward, and have to be performed manually. Composing data flows into compound flows is an example of an abstraction type which is user determined. It is generally impossible to give a priori specifications of procedures on how to compose flows into compound flows that will give meaningful and useful results for all Data Flow models. E.g., sometimes it is desirable to compose all possible compound flows (without combining processes), and sometimes it is desirable to simplify only parts of the model through the use of compound flows.

On the other hand, hiding of data flows, hiding of data stores, and hiding of external entities are examples of abstraction types which may be performed automatically. These abstraction types are applied in full or never at all, they produce results which are fully predictable, and can thus be performed automatically. Applying the reverse of the transformations applied in an “automatable” abstraction to an abstracted model that have not been modified will always produce the non-abstracted model.

As an example of a user determined abstraction, consider the abstraction of the decomposition of process *Estimate Future Need* depicted in figure 6-
6. Obviously, the way the flows have been composed into compound flows is subject to individual preferences. Figure 6-16 depicts a different way of forming the compound flows which demonstrates that assistance from the user is necessary when these types of abstractions are formed.

When a development object has been checked out by several transactions in the form of an abstraction it is clearly advantageous to the communication between transactions that the abstractions checked out to the respective transactions are of the same abstraction type. This also holds true for the final merging process that is to take place. In the case of user determined abstraction types, this can not be guarantied. This makes it feasible to have official abstractions for each abstraction type of the various versions of development objects.

![Diagram](image)

**Fig. 6-16: Estimate Future Need 1.0<2>**

**Different types of abstractions have different scope of effect**

The different types of abstractions applied to a development object can be compared with respect to the scope of effect of the abstractions. This scope of effect to a development object can be divided into two categories: (1) Abstractions where the scope of effect is limited to the development object itself. (2) Abstractions where the scope of effect includes the development object itself, the development objects that refer to the object in question, as well as objects referred to by the referring object in question.

The three abstractions concerned with hiding of external entities, hiding of data flows, and hiding of data stores all have a scope of effect which is limited to the diagram itself, i.e. the development object in question. The abstractions do not affect the development objects that represent the decomposition of any of the processes in the diagram nor development objects which refer to the abstracted development object by a solved by relation. This means that if one applies an abstraction consisting of e.g., hiding of data stores, to a development object of type Data Flow diagram, the abstraction will not influence any related objects like decompositions of
processes. Thus, the abstraction can be defined in terms of components local to the development object.

However, if we consider the abstraction type consisting of forming compound flows, the result is different. The example shown in figures 6-3, 6-6, and 6-9 clearly shows that the compound flows used in figure 6-3 are inconsistent with the compound flows used in the decomposition of process P1, depicted in figure 6-6. (E.g., the definitions of CF-1 used in figure 6-3 is different from the definition of CF-1 used in figure 6-6, as can be seen by comparing the definitions given in figures 6-4 and 6-7.) The inconsistency can be attributed to the mismatch between the flows into process P1, and the flows used inside the process. This means that the definition of a compound flow may influence the flows, compound or simple, in decompositions of processes that are related by the solved by relation. The inconsistency between the two development objects can be handled by allowing compound flows in a decomposition to be defined in terms of simple or compound flows in the parent development component (diagram), i.e. refer to flows in the parent diagram. Thus, a compound flow flowing into a process need not be used as the same compound flow within the development object that represents the decomposition of that process. The input compound flow may be decomposed on the process border and its constituents used as simple flows, or used to form other compound flows that are perceived as more suitable within the decomposition.

| type(CF-1): | Compound Flow |
| components(CF-1): | Replenish Part, Order History |
| connections(CF-2): | RS.Stock Clerk.Replenish Part → CF-1, |
| | RS.D2.Order History → CF-1, |
| | CF-1 → P1.2 |

Fig. 6-17: CF-1 defined in terms of simple flows in Replenish Stock

| type(CF-1): | Compound Flow |
| components(CF-1): | Replenish Part, Order History |
| connections(CF-2): | RS.Stock Clerk.Replenish Part → CF-1, |
| | RS.CF-1.Order History → CF-1, |
| | CF-1 → P1.2 |

Fig. 6-18: CF-1 defined in terms of compound flows in Replenish Stock

In our example, the definition of the compound flow CF-1 that was used in development object Estimate Future Need in figure 6-6, and defined in figure 6-7, may then be expanded to include references to simple flows in the Replenish Stock object (figure 6-3), as depicted in figure 6-17. Alternatively we may chose to expand the definition of CF-1 to include references to compound flows in the Replenish Stock object, as depicted in figure 6-18. When referencing components (flows) in the parent diagram (i.e. the Replenish Stock development object), we have used RS as a shorthand no-
tation for Replenish Stock. Thus, the compound flow, $CF \cdot 1$, flowing into process P1, labelled (Replenish Stock 1.2<1>) $CF \cdot 1$ in figure 6-6 with corresponding definition in figure 6-4, is called RS.$CF \cdot 1$ for short.

6.3. Modifications of abstractions

Modifications of abstractions are to a large extent very similar to ordinary modifications. However, some differences exist since abstractions may trigger changes to the layout of the object, a transaction may consider working on different abstractions of the same object, and merging of abstractions needs some extra care.

Revisions of abstractions

Since the complete (i.e. fully detailed) specifications of an information system will be too complicated to understand, abstractions will be extensively used during development transactions. Abstractions will usually be the objects of revision. Hence, a chain of revisions, local to the development transaction, may originate from the abstraction. This chain will reflect the development work going on as the modification process progresses.

Figure 6-19 depicts a situation where a transaction have checked out a development object, $S \ 1.1$. In the general case, the checked out development object is identified by the name $S \ 1.1\{\tau\}<\alpha>,\lambda$, where $\{\tau\}$ indicates that it belongs to transaction $\tau$, $<\alpha>$ indicates that it is an abstraction of type $\alpha$, and $\lambda$ is the revision number local to the transaction.

![Diagram](image)

Fig. 6-19: Transaction based on abstraction

In this case the first local revision, i.e. the copy that has been checked out to the transaction, is given the name $S \ 1.1\{1\}.0$, signifying that the object belongs to transaction 1, is not an abstraction (due to the lack of an abstraction identifier), and has local revision number 0. Due to the complexity of the development object that has been checked out, we assume that the developer responsible for the transaction decides that further development work is best performed on an abstracted version of the object. An
abstraction is thus applied to this development object (the input object to the abstraction process is the first local revision within the transaction), yielding a new local revision called \( S 1.1(1) \textless 1 \textgreater 1 \).

This local revision is subject to subsequent changes, resulting in a number of new local revisions, the last of which is called \( S 1.1(1) \textless 1 \textgreater n \). At this point in time, the responsible developer is satisfied with the modifications performed, and decides to check the object back into the repository. Before this can be done the development object will have to be brought back to being an object with all the details in the specification. This means going through all of the operations that were executed to perform the abstraction, and reintroducing the details that were hidden by these operations. For any components that were introduced or changed during the local revisions of the abstracted object, details must be added as necessary. The result of bringing the object back to a fully detailed specification is called \( S 1.1(1) \textless n + 1 \). This local revision of the development object can then be checked back into the repository to form version 1.2 of the object.

**Abstractions may trigger changes to the layout**

The systems development methods that are discussed in this text facilitate executable specifications or generation of operational code. These specifications will to a large extent be comprised of diagrams. The diagrammatic representation may be considered as a one-to-one model of the operational system. When an abstraction of a diagrammatic development object has been created, components in the diagram have been removed. Removing components in a diagram may remove spatial constraints on the remaining components in the object. The remaining components may thus in many cases be re-arranged to improve the layout of the abstracted development object. A layout change may serve to improve the way the diagram may be used as a means of understanding and communicating the system structure and behaviour to analysts, designers, and end users.

The changes to the layout are applied to the abstracted development objects when the graphical layout of the object needs re-arranging to further reduce the complexity of the model, thus making the model appear more tidy. Several observations related to the change in layout may be made:

- After an abstraction has been applied it may in many cases be desirable to apply a change in layout.
- Arriving at a new layout is an entirely manual process.
- Returning to the old layout may be done semi-automatically. For components in the development object that have remained unchanged, the old layout may be applied directly. For new components and components that have been changed, a layout must be given manually.
- It may happen that the development of a development object is continued through several consecutive transactions. As it is often desirable to use the abstractions applied in one transaction as the basis for the work in the next transaction, it may also be desirable to use the layout changes that was in effect (if any at all) in the first transaction, in the next transaction.
The natural way to interpret a change of layout is to consider it a modification of the development object just like any other modification. Having changed the object, the result is considered a revision of the abstraction in question. I.e., changing the layout constitutes a local revision of the abstracted development object.

To be able to change the layout, make a number of modifications, and then return to the old layout, the old layout has to be recorded. When attempting to re-apply the old layout to a modified development object, each component will be checked and the old layout will be applied wherever possible. I.e., we use two different layouts, the old and the new layout, for the same development object. This means that one must be able to distinguish between the components of a development object and the layout assigned to each component. The consequence of this requirement is that the layout of the development object is explicitly separated from the content of the object. That is, for each development object, there may be several associated layouts.

There are two ways to present this (internal functionality) to the developer:

- One may either utilise this property in full and explicitly show that an object may have several layouts associated. This means that a specific choice of layout combined with the object would then comprise a version of the abstraction. An abstracted object may then have its layout changed, and changed back, in a conceptually quite simple way; just apply one or the other layout. This solution requires that the representation of the development object maintains a clean separation between layout and content.

- Alternatively, one may allow changes in layout which to the developer is shown as a special kind of local revision. When reintroducing the old layout, this is considered yet another special local revision. This alternative may use precisely the same internal divide between layout and content as the first alternative. It may also be realised without the flexibility of a full separation of layout and content, provided it is possible to separate components from layout in the storage structure.

The rationale behind the decision to allow a layout change in the first place, was to remove undesirable effects on the layout that originated from performing an abstraction. The discussion uncovered no need to support a large number of layouts per abstraction. However, regardless of the alternative chosen, we need to maintain at least two different layouts; the original layout from before the layout was changed, and the new layout that was created after the layout was changed. The original layout would mostly be used on the development object with all the details, while the new layout is used on the abstraction of the development object.

We also need to distinguish a normal local revision, i.e. a local revision where the content of the object is modified, from a local revision which consists of a change in the layout. Provided we permit only one change in layout per abstraction, we may introduce a special local revision signifying a change in layout. When this special local revision is applied, a new layout is given to the development object. The original layout is recorded and reintroduced on the version of the development object that follows the next occurrence of this special local revision, i.e. normally immediately
before one is about to return from an abstracted view of the object to a version of the object with all the details.

Figure 6-20 depicts a chain of revisions based on an abstraction of version 1.1 of development object S. The first local revision of the development object is the application of the abstraction. The second local revision is the change of layout. The resulting component is called S 1.1(1)<1>.2(L). The (L) indicates that this local version is the result of a special local revision which has changed the layout of the component and that no functional modifications have taken place in going from S 1.1(1)<1>.1 to S 1.1(1)<1>.2(L).

Once the new layout has been established, the development will take place using that layout. In the example the development is continued until local version S 1.1(1)<1>.n(L) has been developed. At this point the developer decides to check the object back into the repository. Two steps are necessary to be able to do this:

Firstly, the old layout must be reintroduced. In the example the result of this operation is called S 1.1(1)<1>.n+1. In the case that the development transaction have resulted in adding new components to the abstracted development object, i.e. modification of the content in the form of new components, their position in the original layout (in which the new components originally did not exist,) can not be determined from the old layout. New components may have to be given a layout manually as part of the process of reintroducing the old layout. It may also be possible to have the support system make suggestions on the placing of new components based on heuristics for graph layout, minimising the number of lines crossing, etc..

![Fig. 6-20: Layout transformation](image-url)
Secondly, the object must be brought back to a fully detailed specification. The result of this operation is called $S 1.1(1)<1>.n+2$. This local version may now be checked back into the repository.

The solution where one implements a separation of layout and content would require significant modifications to all contemporary CASE tools. The tool would have to be augmented with provisions for making the distinction between content and layout explicit. Additional functionality to properly utilise this distinction would also have to be developed. This amounts to quite substantial changes to a tool. Should the changes be feasible to implement they would have to match the tool architecture. Problems concerning incompatibility between the required modifications and the tool architecture would disappear in a situation where one is engineering a new CASE tool rather than modifying an existing one. The architecture of the new tool may then be tailored to take full advantage of the split between layout and content, and providing the necessary functionality would thus be less costly.

**Abstractions developed in parallel**

A particular abstraction of a development object reflects one particular way of looking at this object. The abstraction may be characterised by certain components of the development object that has been highlighted by removing other elements. It is possible to define several different abstractions, reflecting different ways of looking at the same development object. The different abstractions may be useful for different purposes.

Thus, it may be that within the same development transaction, it is considered useful to work on a development object using several different abstractions (of different types) of that object. The development based on different abstractions will result in parallel chains of local revisions of different abstractions of the same development object.

![Diagram](image)

*Fig. 6-21: Parallel abstractions within a transaction*

Figure 6-13 depicted three different abstractions of *Replenish Stock 1.2*. Imagine now that the developer responsible for the object would like to
make certain modifications to the object. The implementation of the modifications will comprise the development of a new version, version 1.3, of the development object. In order to do this he checks out version 1.2 of the object, and decides to use three different abstractions as the basis for the development. Figure 6-21 depicts a situation where development transaction 1 has checked out version 1.2 of the Replenish Stock development object. Three different abstractions of the development object have been established.

A development object is named $RS\ 1.2\{\alpha\}<\alpha>\cdot\lambda$, where $\{\alpha\}$ identifies the transaction, the $<\alpha>$ identifies the different abstractions, and the $\lambda$ identifies the local revisions for each abstraction. Thus, one of the development objects is named $RS\ 1.2\{1\}<2>\cdot 2$, indicating that it belongs to transaction 1, is of abstraction type 2, and has local revision number 2.

Each abstraction is developed separately. Thus, each abstraction will be the basis for a chain of local revisions. The development continues until the developer responsible decides to check the object back into the repository. At this time there are two possible situations:

Either the development work during the transaction turned out to use just one of the abstractions. Consequently, only that chain of revisions have been developed properly. In this case the development of the other abstractions may have been used to investigate certain problems, and then discontinued. The results of the investigations may subsequently have been transferred to, or used in, other revision chains. The other abstraction types may also have proved less fruitful than anticipated. In this case they would not be further developed at all.

Fig. 6-22: Parallel abstractions checked in
The last local version of the abstraction that turned out to be used may then be brought back to a fully detailed specification, and the development object checked back into the repository.

Alternatively, the development transaction produced two or more, equally developed chains of revisions. In this case the resulting local versions of two or more development objects based on different abstraction types will have to be merged before check-in of the development object.

Figure 6-22 depicts a situation where abstractions of type 1 and 3 have produced local versions which needs to be merged. Abstraction type 2 was abolished after local version RS 1.2(1)<2>.m was developed. The two local versions which are going to be merged are RS 1.2(1)<1>.k and RS 1.2(1)<3>.n. Development objects that are of different abstraction types, will be implemented using languages that have different expressiveness. The objects may therefore not be compared directly and consequently, they may not be merged. Before merging may be done the development objects will have to be brought back to full detail. This results in the creation of development objects RS 1.2(1)<1>.k+1 and RS 1.2(1)<3>.n+1 who in turn are merged to produce RS 1.2(1).p which may be checked into the repository.

When the abstractions were brought back to full detail in figure 6-22, the abstraction identifiers were retained. This was done even though the local versions no longer were abstractions. The reason this had to be done was to be able to distinguish between the different local versions, and the fact that they originated from different abstractions. Another way of solving this situation would be to use nested transactions. At the time the abstractions were applied, one would initiate three sub transactions, one for each abstraction type. The subsequent merging would then be performed with local versions from three different sub transactions as input.

Merging of abstractions

As can be seen from the previous section, the consequence of allowing several parallel abstractions is the need to merge the different abstractions immediately before check-in of the object. Merging development objects based on different abstractions within a development transaction is completely analogous to merging development objects based on different abstractions from different transactions. However, merging of different abstractions that have been developed within a development transaction will have to be done before the transaction is committed, i.e. before any merging with development objects that originate from other transactions is to take place.

The merging can be done in two ways:

If the development objects that are going to be merged are of the same abstraction type, then the abstractions may be merged before returning to fully detailed specifications, i.e. they may be merged as abstractions.

If the development objects have incompatible abstraction types, then the objects must be merged using fully detailed specifications. That is, one must bring the objects back to fully detailed specifications before merging can be done.
The strategy used for the merging itself is identical irrespective of whether the development objects are abstractions or objects with fully detailed specifications.

In the case that the layout have been changed on any of the objects that are going to be merged, two problems arise: What layout to use during the merging operation? And what layout to use on the merged version.

When a development transaction is about to terminate, and new versions of the development objects checked out to the transaction is about to be established, one is about to return to the same level of expressiveness and detail in the specifications as held by the previous version of this development object. Thus, any changes in layout that were applied during the transaction will be less appropriate to apply on the merged development object. To use the original layout as the basis is far more appropriate. The consequence of this is that it is more appropriate to return to the old layout before merging starts. This will ensure a merged development object with the minimum difference in layout to the previous version of the object.

Fig. 6.23: PHM model for development transaction
6.4. Conceptual model of development transactions

To summarise the discussions on management of change and change support for complex models, a conceptual model of the development transaction is depicted in figure 6-23. To reduce the complexity of the figure, we have taken two steps; we have omitted the influence of versioning in the diagram, and we have omitted a number of connections which we feel are of less importance than the ones left in.

The diagram illustrates several points:

- There are two kinds of development objects; frozen development objects and placeholder objects. The frozen development objects comprise all committed versions of development objects, i.e. objects that have been checked into the repository as public and as such are immutable. The placeholder objects comprise versions of objects that are in development, i.e. checked out into a development transaction. Placeholder objects are sometime in the future filled by a local development object.

- Local development objects have been checked out into a development transaction, and as such are owned by that transaction. There are two kinds of local development objects; full local development objects and abstracted local development objects. The full local development objects are objects with full specificational detail. The abstracted local development objects are objects where some specificational detail have been abstracted away. A local development object may be the resolvent of a merging operation.

6.5. Chapter summary

Work on complexity reduction techniques in information system modelling is a relatively new research subject. We believe that this subject is of growing importance as the expressiveness of information system models grows. This chapter has provided a discussion of abstractions as an example of one important complexity reduction technique. Version and configuration management mechanisms to support abstractions have been proposed, and the influence of the use of abstractions on development transactions has been discussed. These issues comprise the necessary augmentations to the cooperative work management framework.

The discussions in this chapter has contributed to the validation of the framework for cooperative work management that was developed in the previous chapter.

However, there are still some improvements to be done and issues to be discussed:

- The names of the various local versions may be quite complex. This is due to the requirement for unique naming of development objects. To simplify the names, it is possible to use aliases for parts of the name. An alias may easily be expanded, thus giving a full unique name. This may make it simpler to work within a transaction when the context of the work is mostly in the local sphere.

- The proposed framework records and maintains various relationships among development objects, e.g., revision, variant, check-out,
check-in, abstraction, layout. It is possible to have attributes associated with each of these relationships. The attributes may allow automatic and semi-automatic reasoning on the relationships, e.g., recording of design decisions. The precise contents and semantics of the attributes depend on what information one wants to record, and on what kind of reasoning one wants to do.

- Project management considerations require that one is able to estimate the development state of the system and of its components. The value of the state attribute assigned to each component is a project specific estimate of the development state of the component. The possible values may be chosen to give the project quality measures that match the project specific requirements. E.g., the quality states defined by Lindland may serve as an illustration of one such set of states. The states are illegal, conformant, consistent, unambiguous, valid, complete, and optimal [99].
7. Management of change for a CASE tool repository

This chapter applies the framework for cooperative work management that was developed in chapters 5 and 6 to build a repository solution for an experimental integrated environment for information system development called PPP (Phenomena, Processes, and Programs). Firstly, the chapter gives an introduction to PPP. Secondly, requirements to a repository solution for PPP are developed. Thirdly, alternative integration strategies are discussed and a repository architecture is proposed. Finally, a repository schema design is developed.

7.1. Introduction to PPP

The PPP (Phenomena, Processes, and Programs) is an experimental integrated CASE tool for software development [62]. PPP was developed at The Norwegian Institute of Technology by Prof. Arne Sølvberg and his students.

The development of PPP was motivated by methodological deficiencies in contemporary CASE technology. Two of the most prominent being the lack of formality in the specifications developed, and the existing information gaps between different parts of the specifications in contemporary CASE tools. The PPP is intended for use in all stages of information systems development. Hence, the PPP has a two faceted focus in trying to provide increased:

- Formality in the specifications. Formality facilitates early verification and validation of specifications, as well as automated code-generation.
- Integration between the different modelling languages used within the framework, specifically between different languages used during different stages of development.

Clearly, introducing increased formality and model integration will increase the complexity of the formalism. This puts added strength to the need for tool support.

The tools

The PPP tool comprises four strongly interrelated diagrammatic modelling tools and five specification manipulation or specification transformation functions, figure 7-1.

The diagrammatic modelling tools model both dynamic and static aspects of the real world, including detailed algorithmic descriptions of functionality as well as user interfaces.

**The process modelling tool**, PrM, supports a formalised extension of ordinary Data Flow diagram modelling used to represent functional decomposition structures.
The phenomenon modelling tool, PhM, supports a formalised extension of Entity Relationship diagrams used to represent static objects.

The process life description tool, PLD, supports low-level diagrammatic representation of algorithms.

The user interface description tool, UID, supports low-level definition of screens.

The specification manipulation and specification transformation functions operate on specifications in work spaces developed by use of the diagrammatic modelling tools.

Verification functions, support consistency checking of PPP models, both within one language and between different modelling languages.

Validation functions, support validation of specifications through model execution and explanation generation functions.

Code generation functions, support generation of executable code from the PPP specifications. Generators have been developed for ADA, SIMULA, TEQUEL (a temporal logic rule language), and C.

Report generation functions, support generation of documentation of various facets of PPP models. Information is automatically extracted from a PPP model in a work space.

The save/load functions, load a model from a UNIX file into a workspace and save the contents of a workspace into a file.

---

Fig. 7-1: The PPP system architecture [62]
The method

The PPP advocates a top-down, incremental and iterative development method, figure 7-2. Development starts with establishing a conceptual model of the real world system in question. This conceptual model is also called a real-world model. The real-world model describes both the existing manual information system and its environment. The model is iteratively refined until a suitable level of detail is established. Following this iterative phase, it is decided which part of the real-world organisation which will be automated. This is expressed using automation boundaries on the refined real-world model. The part of the model inside the automation boundaries becomes the first, abstract specification of the computerised information system.

![Diagram of PPP information systems engineering method](image)

Fig. 7-2: The PPP information systems engineering method [62]

The PPP approach perceives development of computerised information systems as a mapping between a manual system and a system supported by computers. Thus, a functional analysis of the manual system also covers a design of the computerised system. Therefore, once the automation boundaries has been determined, the focus on further refinement of the part of the real-world model that is inside the automation boundaries shifts from being primarily analysis oriented to being design oriented. Iterated decomposition of the computerised part of the information system continues until the specifications reach a level of formality from which executable code for the target system can be automatically generated.
The language

The PPP language is formed by integrating four basically independent sub-languages. The sub-languages are based on widely accepted modelling approaches that have been modified to meet the two focal points that were identified for PPP; increased formality and model integration.

The PPP language is suitable for representing a wide range of real and artificial systems. I.e., following the PPP approach it is possible to model the information system's internal architecture, the communication between parts of the information system or between information systems and its environment, and even the environment of the information system.

In a PPP model the system is represented by means of four basic concepts:

**Static objects** Static objects denote natural or artificial objects that characterise important static aspects of the information system and its environment. To specify those objects and their interrelations, the Phenomenon Model, PhM, is used.

**Functional structures** Functional structures show the decomposition of systems into subsystems, the interaction between subsystems, and the transformation of data in a particular system or subsystem. They document the internal functional architecture of a system. I.e., the structures explain the functions of a system, and describe how these functions are constructed by combining other functions. The Process Model, PrM, is employed to form functional structures in PPP.

**Algorithms** The overall dynamic structure of the information system is described using a hierarchy of functional structures. When the internal structure of a function is sufficiently simple, an algorithmic description of the function may be appropriate. These descriptions are specified as Process Life Description, PLD, models.

**User interfaces** The specification of user interfaces is ideally independent of programming environment and only partly dependent on the functional structures of the computerised information system. The User Interface Description, UID, model is used for these specifications.

In the rest of this section we will focus on the PhM and PrM sub-languages. The PLD and UID will be omitted from our discussions. We proceed to describe briefly the PhM and PrM.

**PhM - Phenomenon Model**

The Phenomenon Model, PhM [136][138], is based on the classical Entity Relationship model, ER model [25]. The diagrams are used to describe the static properties of the problem domain and the static objects of the information system.

Being based on the classical Entity Relationship model, the PhM language exploits the simplicity and elegance of Entity Relationship diagrams. The notation is somewhat different, but the general structures are the same. To improve the expressive power of the language some additions have been made:
The attribute relations The attribute relations are extended to give a more accurate description of entity and relationship properties. Both entityclasses and relationshipclasses may be related to data types through attribute relations in PhM.

The attribute relations are further characterised by their types:

- **Identifiers** An entityclass must have exactly one identifier attribute. The corresponding data type shows the possible values of the identifier. An instance of an entityclass is uniquely determined by that value. An entityclass cannot have several identifier attributes.

- **Attributes** Ordinary attributes are used to give entityclasses and relationshipclasses some kind of properties. An entityclass or a relationshipclass may have an arbitrary number of attributes. Attribute values are not supposed to be unique within an entityclass. The concept of attribute in the classical Entity Relationship model corresponds to this type of attribute relations.

- **Repeating groups** A property of an entity may be given by a set of values. If these values are taken from the same data type, the property can be modelled as a repeating group. A repeating group associates an entityclass or a relationshipclass with a data type, indicating that the specified property consists of a possibly infinite set of values.

- **Qualities** A quality is an aspect of an entityclass as a whole, and not of particular entity instances. Thus, entity instances may not be quality related to any data type.

**Properties of relationships** In traditional Entity Relationship models the relationships have a cardinality specification. PhM extends the expressive power of relationships by adding a coverage specification. Each entityclass in a relationshipclass is either

- full, meaning that all entity instances of the class must participate in some relationshipclass instance, or

- partial, if this is not necessarily the case.

**Subclasses** Subclasses are used to model subsets of entityclasses or subsets of other subclasses. If an entityclass has a subclass, all instances of the subclass are instances of the entityclass. However, an instance of the entityclass is not necessarily an instance of the subclass.

An entityclass or a subclass may have several subclasses. A subclass, on the other hand, has exactly one superclass.

**PrM - Process Model**

The Process Model, PrM, is based on the Data Flow diagrams from structured analysis and design [55]. The expressive power of the Data Flow diagrams has been improved by adding temporal constructs and a more precise specification of data and control flows.
For a formal definition of the PrM language see [103]. The definition of the PrM language made some additions to the traditional Data Flow diagrams:

**Resources, timers, and agents** Resources and timers are included in PrM, while the external entity concept of Data Flow diagrams is substituted with an agent concept.

- **Resources** A resource contains items that are needed for processes to run. The state of a resource is given by the number of these items. The items themselves are only useful due to their existence, and are not associated with any information aspect. They are added to or removed from a resource by the processes connected to that particular resource.

- **Timers** Timers extend the temporal expressive power of PrM. There are two kinds of timers: clocks and delays. Clocks are used to model events that are to occur at specific moments in time. Input flows to the clock activate or deactivate the clock, while an output flow denotes the specified clock signal. Delays are used to model events that are delayed with respect to time. They have at least two flows, one input and one output flow. A real value associated with the delay determines the time period between arrival of input signal and sending of delay signal.

- **Agents** Agents are used instead of external entities, emphasising the dynamic properties of the entities. There is no semantic difference.

**Auxiliary concepts** The auxiliary concepts enable the representation of control flows and logical relations between inputs and outputs:

- **Triggering and termination** The notions of triggering and termination makes it possible to introduce the modelling of control flow. An input flow that is triggering can only be received when a process is not active. The receipt of a triggering flow will cause a process to change its state from idle to active. Similarly, the sending of a terminating flow will cause a process to change its state from active to idle. The triggering and terminating flows thus decide when the process will start and stop. However, non-triggering and non-terminating flows may be sent at any time while the process is active.

- **Ports** Input and output ports are used to define logical relations between inputs or outputs. The symbols are the same both for input and output ports. While input ports are placed inside the process, the output ports are placed outside. There are three basic kinds of ports:
  - **AND** The AND port is interpreted such that all member flows, or flow groups, are going to be received or sent.
  - **XOR** The XOR port is interpreted such that only one member flow, or flow group, is going to be received or sent.
  - **OR** The OR port is interpreted such that one or more of the member flows, or flow groups, are going to be received or sent.

Moreover, a port may have the additional property of being:
Conditional A conditional flow is interpreted such that the flows, or flow groups, within it need not be received or sent.

Repeating A repeating flow is interpreted such that the flows, or flow groups, within it will be received or sent several times during one process execution.

Composite ports may be constructed by putting ports inside each other.

The details of a meta model for PPP is given in appendix E.

The PPP storage structure

The current PPP tool implementation stores the specifications in Prolog fact files. The fact structure is divided into facts for overall system structure, facts for PhM diagrams, and facts for PrM diagrams. The details of the fact structure is given in appendix F. The fact structure reported has been designed to implement the meta model that was presented in appendix E.

![Partitioning of the PPP Facts](image)

In an orthogonal view of the fact structure it may be seen as partitioned into a layout part and a contents part. The contents part can be further partitioned into structure information and other information, see figure 7-3.

7.2. Requirements to a repository for PPP

A repository for a CASE tool is a database in which to store the various models that are being created by use of the tool. Several requirements apply to a repository solution: Firstly, there will be requirements to the repository that apply because the repository is just another database. Secondly, there will be requirements that apply because the repository is a database for the CASE tool PPP. Thirdly, there will be requirements that apply because we want to implement a certain functionality for cooperative work management, CWM. Hence, building a repository for PPP is is the same as building a database to store the specifications created by use
of the PPP CASE tool. The PPP Repository will act as the central repository of specifications for multiple PPP users.

The repository is a database
The PPP Repository is the database for the PPP CASE tool, and is thus required to exhibit most of the features known to be characteristic of a regular database management system. The majority of these requirements apply regardless of the technology chosen to implement the repository [65]:

- Non-redundant data storage to avoid unnecessary duplication of the same data.
- High-level data access to ensure that data access procedures are handled by the DBMS and not written into each application.
- Data independence to avoid changing the application when the physical storage is rearranged.
- Openness to allow new applications to access existing data.
- Query facilities to provide a simple user interface for interactive queries.
- Report generator to allow the user to easily define standard reports.
- Real-time updating ensures that the last state of the data is always available.
- Locking prevents simultaneous attempts to update the same object.
- Concurrency ensures that multiple tools and multiple users may work concurrently on "live" data.
- Integrity rules defining acceptable states of the database may be enforced by the DBMS rather than by application programs.
- Security ensures that objects are protected from unauthorised reading, writing, or deletion.
- Recovery ensures that the database is restored to the last consistent state after an abnormal event.

Requirements from the PPP CASE tool
The PPP CASE tool put special requirements on the database support. Some of the items mentioned above will be given a special interpretation in a CASE tool context:

- Real-time updating ensures that the last state of the data is always available.
  
  Real-time updating is useful to anyone who wishes to know the current state of the database, but of questionable value to systems developers, who need to work against a stable set of development objects. The PPP repository is required to offer stable development objects that can been checked out into private work spaces. For this purpose locking may be helpful.
- Locking prevents simultaneous attempts to update the same object.
Locking is useful to prevent inconsistent updates when the duration of the typical transaction is short. For CASE tools, the typical duration of a design transaction is in the order of days and weeks, rather than seconds. Locking in the traditional DBMS way is therefore not feasible. The PPP repository is required to offer a relaxed locking concept where many transactions may check out the same object from the repository. Objects that are in a checked out state are prevented from being updated by anyone but the development transactions that have checked out the object. Check-in of an object, thus updating the repository, is only allowed following a consensus among the transactions that have checked out the object.

- Concurrency ensures that multiple tools and multiple users may work concurrently on “live” data.

The concept of “live” data does not apply to CASE tools and repositories in the same way as to information systems and DBMSs. It would be unusual for two developers to work simultaneously on the same working set of data, i.e. the same copy of the development objects. The PPP repository is required to offer developers the possibility to work on copies of the same development objects in different work spaces. The copies would then be used as input in a merging process where the purpose is to achieve consensus among the developers involved in working on the object.

- Integrity rules defining acceptable states of the database may be enforced by the DBMS rather than by application programs.

For PPP the integrity rules are defined from the rules of the PPP models and methods. Enforcement of integrity rules that apply to development objects that are currently checked out, would normally be done during the development transactions. Those integrity rules that concern the relationship between the objects in the repository and the objects that are currently checked out would have to be enforced by a cooperative work management system, i.e. the layer on top of the database. Care must be taken when enforcing the integrity rules since the rules which would apply to a finished set of development objects are inevitably violated by work in progress. Hence, neither the PPP tool nor the repository must be too rigorous too soon.

Requirements from CWM software

To implement the cooperative work management functionality presented in chapter 5, we briefly summarise the items that need to be satisfied:

- Representation of generic structures, hierarchical and flat.
- Versioning of development objects.
- Support for nested development transactions.
- Relaxed authorisation rights to support parallel work.
- Synchronisation among transactions through a publish and subscribe mechanism.
- Negotiations among transactions, e.g., merging of objects, to arrive at consensus.
Realising a PPP repository

PPP stores Prolog facts in standard UNIX files in a prototype implementation from 1991. All Prolog facts comprising a diagram in PPP is stored in one file. This means that each scenario in a PhM model is stored in a separate file, each process diagram (each decomposition is a separate diagram) in a PrM model is stored in a separate file, each PLD diagram that gives an algorithmic definition of a leaf node process is stored in a separate file, and each UID specification is stored in a separate file.

Every time the user updates his model, the PPP tool updates the file where the relevant Prolog facts are stored. If the user decides to keep several versions of the model, one file per version is required. If several users cooperate on the development of related models (diagrams) separate files for each user will have to be created and maintained manually.

All aspects of versioning of development objects and all synchronisation among multiple users is manually managed by the developers.

When we implement cooperative work management functionality in PPP we require a tight integration between the cooperative work management functionality and the tool. There are two different alternatives to integrate this functionality into the tool:

- One can either add the functionality to the tool itself. This requires significant modifications to the tool.
- Alternatively, one may interface a standalone cooperative work management front-end, CWM front-end, to the PPP. The latter alternative has two advantages: (1) There will be less modifications to the PPP, and (2) a generic cooperative work management front-end will be developed. A generic cooperative work management front-end may be interfaced to several tools.

7.3. Implementing cooperative work management functionality

The tight integration of CWM functionality with the PPP CASE tool must satisfy the following requirements:

- The PPP tool must have control of the way the CWM functionality operates. I.e., the user dialogue with the CWM sub-system is initiated and controlled by the CASE tool.
- The target system structure developed through the use of the CASE tool is stored using the CWM sub-system. This means that it must be possible to implement the meta model of the CASE tool in the CWM sub-system.

7.3.1. Alternative integration strategies

Recalling from the previous section two alternatives to integrating cooperative work management functionality with PPP were identified: (1)
Adding the CWM functionality to the tool itself. (2) Interface a standalone cooperative work management front-end, CWM front-end, to the PPP.

While the former alternative requires significant modifications to the tool, the latter alternative requires less modifications to the tool, and leads to the development of a cooperative work management front-end that is generic. A generic cooperative work management front-end may be interfaced to several tools.

We proceed to investigate each alternative separately.

**Adding CWM functionality to the PPP tool**

The PPP tool was never designed to cater for cooperative work management requirements. Consequently, the existing fact structure was never equipped with facts which will adequately handle these requirements. As described in appendix F, facts exist to record and manipulate the PPP models on a single user single version basis. Facts to handle CWM functionality are missing. Thus, introducing CWM requires the existing fact structure to be augmented. This has implications on the database schema for the repository.

When the PPP tool is expanded through the integration of CWM functionality, additional data about the development object has to be stored. This means that the complete meta model of PPP will have to change to incorporate the information necessary to realise CWM functionality. As a consequence of this, the database schema of the tool and also the tool itself will have to change to some extent to reflect these changes. One must therefore decide how to add on the information necessary to be able to support CWM.

![Diagram](image)

*Fig. 7-4: Integrating CWM into the PPP architecture*
There are two alternatives for expanding the existing meta model such that it can cope with the cooperative work management requirements:

- One can let the necessary CWM constructs be known to the existing PPP tool. This means that one adds the necessary concepts to handle the configuration management to the existing PPP meta model (database schema). Thus, dramatically changing the existing meta model. The PPP tool will have to change accordingly.

- Alternatively, one can try to hide the introduction of constructs necessary to provide CWM functionality from the PPP tool as much as possible. This means to leave the existing meta model (the one that the current PPP tool knows) as much as possible without changes, and add a functional layer which provides and manipulates the necessary concepts to handle the CWM. All access to development objects will be through the CWM layer which will provide the structure, the versioning, the multi user synchronisation, and the persistent storage. This means that the PPP will be allowed to operate against an almost unchanged meta model.

Clearly, the amount of work that will have to be done with respect to rewriting the existing PPP tool, is less if one choses the latter alternative. Figure 7-4 schematically depicts the differences between the current solution (left) and the latter design alternative (right).

A generic CWM front-end

Interfacing a standalone CWM front-end to PPP requires the PPP to work on checked-out objects in a work-space. The CWM front-end will provide the objects in private work-spaces and control the versioning and multi user aspects of the development.

The CWM component handles product structure information, versioning information, multi user synchronisation, and storage of temporal and permanent information (both process and product).

The CWM front-end provides a multi user interface to a generic versioned structure of objects that offers an improved transition between synchronous and asynchronous modes of working. An interface to, and the operation of, an object store (DBMS) is provided.

The generic CWM front-end will have the following properties:

- The CWM front-end system will have a separate graphical user interface for stand-alone use.

- The CWM front-end will have an API (application programming interface) which allows it to be called from any application, including the PPP ICASE tool.

- The CWM front-end system can handle generic object structures of two types; hierarchical and flat, and can represent and maintain relations between the two.

- The CWM front-end system stores objects in a database system according to the generic structures defined by the meta model of the
PPP languages. The instantiation of the generic structures into specific structures is done by the user as he develops his models.

Figure 7-5 depicts the PPP architecture where the original PPP tool is interfaced to a generic CWM front-end. This front-end comprise the repository for PPP. The front-end consists of a set of modules to realise the multi user multi version functionality and a DBMS to realise persistent storage. The CWM component offers functionality to check-out and check-in development objects within the context of development transactions, handle the synchronisation of multiple users, etc..

The attraction of this solution is that it involves a minimal number of changes to the existing PPP, while at the same time utilising a generic solution to the repository component.

![Diagram](image)

Fig. 7-5: The PPP with a generic CWM front-end

### 7.3.2. Architecture of the chosen solution

The generic front-end solution is the preferred solution.

The work-space layer in the PPP tool may still operate on Prolog facts with the same structure as before the introduction of the CWM. Hence, the original PPP fact structure may remain stable. The fact structure contains Prolog predicates that keeps all information necessary to manipulate PPP models. It is divided into layout related predicates and content related predicates. The content related predicates includes predicates that record the structure of the models being built. The CWM component must therefore analyse the predicates that records the structure to build and populate the structure that represent the models being built and stored by the CWM.
When checking out a development object, the CWM component reads the database, strips off the additional information required by the CWM system relative to the original Prolog fact format, and provides the PPP tool with the relevant set of Prolog facts in a PPP work-space. This means that the PPP tool will be able to operate against a data structure which is identical to the datastructure that existed before it was integrated with the multi user versioning repository.  

The CWM front-end will be activated from PPP whenever the user wants to work on some development object. The CWM system handles navigation in the object structure, browsing of development objects, and check-out and check-in of objects between multiple users. The objects that are going to be modified by PPP, i.e. objects that have been checked out, are labelled appropriately and transferred to the user’s PPP work-space.  

A refined architecture is depicted in figure 7-6.  

The architecture comprises modules of three categories: Modules belonging to the original PPP tool, modules belonging to the CMW front-end, and modules added to the PPP tool to realise the tight integration between the PPP tool and the CWM front-end.  

![Figure 7-6: Refined PPP + CWM architecture](image)

The Global Model Management User Interface part of the PPP user interface initiates calls to the Common User Interface API to facilitate navigation in the object structures, check-out, check-in, subscriptions, etc. A global check-out places the object in a local workspace belonging to the user.  

The Local Model Management User Interface part of the PPP user interface initiates calls to the Common User Interface API to facilitate navigation, local check-out, local check-in, etc., among checked out object structures in the local workspace belonging to the user.  

The PPP functions data access comprise the functions necessary to read and write data between the PPP tool cashe and the local workspace belonging to the user. These functions access the CWM database functions di-
rectly. Translation of the data from the storage structures of the DBMS to structures used by the PPP tool is performed here.

7.4. Repository schema design

The repository in the architecture depicted in figure 7-6 will contain development objects from several domains. To arrive at a storage representation of the PPP development objects suitable for implementation with CWM functionality, it is necessary to design a data storage schema for the PPP Repository which will satisfy the requirements from the PPP tool perspective summarised in the previous sections.

7.4.1. Objects from several domains

The specification repository collects all relevant development project data. The development project data may be simultaneously accessed by all interested parties in the development process. The repository contains data to serve several needs: The systems analysts build real world models to be able to derive and document requirements. The systems designers develop conceptual designs to satisfy the requirements. The programmers develop programs to realise the conceptual designs. The project manager keeps track of who does what, where, and when.

To be able to satisfy these requirements, the repository must encompass concepts from four different domains: The subject domain, the implementation domain, the development object administration domain, and the development project domain. The names of the domains are inspired by the subject domain, the interaction domain, and the implementation domain of Winograd [155]. The domains are constructed during the course of the development project.

The schema for such a repository would consist of four partly overlapping sub-schemata, i.e., four interrelated sub-models are considered, as illustrated in figure 7-7.

- The subject domain sub-schema
  The concepts in this sub-schema are concerned with the subject domain, and with building a model of the real world that caters for behavioural, performance, and environment interaction aspects. It is important that the model lends itself to transformation into system specifications.

- The implementation domain sub-schema
  The concepts in this sub-schema are concerned with system specifications on various levels of abstraction. The system specifications are related to both the subject domain model and the development object administration model.

- The development object administration domain sub-schema
  The concepts in this sub-schema are concerned with system and system component management. Concepts such as versions and configurations of systems and system components are related to the sys-
tem specifications on one hand, and the development project organisation on the other.

- The development project domain sub-schema

The concepts in this sub-schema are concerned with the development project organisation, monitoring, and management of project resources.

The PPP tool records and maintains data belonging to the subject domain sub-schema and the implementation domain sub-schema. The software for cooperative work management introduces and maintains the concepts in the development object administration domain sub-schema. The functionality necessary to manipulate concepts in the development project domain sub-schema may be added to the tool either by interfacing to a separate tool, or by developing such functionality to be an integral part of the PPP tool.

![Diagram of concepts in systems development]

Fig. 7-7: Concepts in systems development

As suggested by this initial sketch of the repository schema, the repository will contain interrelated sets of concepts partly maintained by the CASE tool, partly by software for cooperative work management, and partly by
project administration software. Precisely which concepts and what the relationships between them are, is determined by the meta models of the CASE tool, the software for cooperative work management, and the project administration software respectively.

### 7.4.2. Choosing development object granularity

The previous section discussed the structure of the schema. According to the discussion the schema should consist of concepts to describe and manipulate four different domains: the subject domain, the implementation domain, the development object administration domain, and the development project domain. We have also discussed the tools, existing and prospected, which will operate against the schema. The meta models of these tools will determine exactly which concepts that will be stored.

One remaining issue regarding the design of the representation of the development objects stored in the repository, is the granularity of the stored concepts. The granularity is determined by the degree to which the meta models of the PPP tool, the software for cooperative work management, and the project management software, are implemented. The granularity may range from a coarse grained solution where a target system from a repository point of view is perceived to consist of relatively few large objects, to a maximally fine grained solution where the object structure in the repository is a one-to-one mapping of the meta models.

A target system that has been developed using a mixture of manual methods and computerised development tools will always be specified by a number of both formalised and non-formalised development objects. The non-formalised development objects are often called documents. The number of documents that has been used relative to the number of development objects that is used is determined by two elements; the systems development method, and the number of computerised development tools available with this method.

The specifications of a target system will consist of a set of documents and a set of development objects.

- A document in this context, is a structured amount of information (text, diagrams, sound, etc.) intended for human perception, and that can be interchanged as a unit between applications and users [157]. A document of this kind is normally developed using a document processing system. Note that our definition of documents do not include “documents” produced using any computerised systems development tools. Documents of this kind are intended as input for other development tools, and are thus categorised as development objects in this context.

- A development object in this context, is a structured amount of information produced by a computerised systems development tool, and intended both for human perception and as input for other computerised systems development tools. A development object is either flat or hierarchical.

The documents may be related to the development objects in an informal way by the use of “pointers” of various kinds. These relationships are motivated by the systems development method. Relationships like “where
used”, “motivates”, “requires”, etc. may provide important help during the systems development process.

Document granularity

The documents may be structured to meet certain standards. The advantages of using a standardised document format are mostly related to exchange and processing of documents. Firstly, documents that are structured according to a standard may be freely exchanged among different document processing systems that support the standard. Secondly, standardised documents lend themselves more easily to automatic processing. Hence, international standards should be used wherever possible. The international standards ODA - Office Document Architecture, ISO 8613 [79], and SGML - Standard Generalised Markup Language, ISO 8879 [80] have achieved international recognition and a number of word processing systems currently support one or the other.

E.g., a document in ODA contains information that is related to its content and structure. The content of a document is partitioned into content portions. Content elements that make up the content portions are governed by appropriate content architectures, e.g., character content architectures or raster graphics content architectures. The rules for representing the structure of documents are collectively called the document architecture. The relationships between the document structures and the content are defined through the interfaces between the document architecture and the content architectures. ODA comprises two levels of abstraction: The structural model which describes in an abstract way the inherent structural characteristics and content of a document, and the descriptive model which describes how the properties of the document are represented.

Documents structured in accordance with ODA may be interchanged using the standardised interchange format ODIF - Office Document Interchange Format. According to ODIF a document is represented by a data stream consisting of standardised interchange data elements. Four different sets of rules to order the interchange data elements within a data stream are defined.

In much the same way, documents structured according to the SGML standard may be interchanged using the standardised interchange format SDIF - SGML Document Interchange Format.

The hierarchical document architecture supported by ODA lends itself easily to implement several levels of granularity. The most coarse grained would leave each document as one object. While the most fine grained would represent each document as a hierarchy of logical objects, with the individual basic logical objects as the leaf nodes in the hierarchy. The techniques developed for versioning of hierarchical development objects is directly applicable to documents.

Development object granularity

The development objects are output from, and input to, computerised development tools. The structure of the objects is determined by the tools and the concepts supported by the tools. More precisely, the structure of the ob-
jects is determined by the way the tools implement the meta models of the systems development method. The degree to which the meta models of the method can be supported by, or implemented in, the repository is therefore determined by the tool.

The granularity of the development objects is the result of a compromise between several factors:

- The storage structure should ideally reflect the structure of the objects manipulated by the PPP tool. This means a simple one-to-one mapping between the tool and the repository. This calls for a storage structure which is fine grained.

- The granularity of the repository has direct impact on the possible functionality in a front-end for cooperative work management which is operating against this repository. A coarse grained repository means limited possibilities for cooperative work management support. E.g., when the same development object is checked out simultaneously by many users a potential conflict situation has occurred. It may not be a real conflict, since modifications may take place in non-overlapping sub-parts of the object. A fine granularity will reduce the probability of multiple users checking out the same object at the same time.

For example, if the repository were to keep data flow diagrams as single objects where all decompositions of processes in the diagram were part of the same object, this would lead to an increase in unnecessary flagging of update conflicts. A fine grained repository means better possibilities for cooperative work management support. Hence, to provide improved support for cooperative work, the object structure should be fine grained.

- Dealing with a large number of interrelated objects is more complex and time consuming than dealing with a few objects. The cost and complexity of handling a fine grained solution requires that measures are taken to arrive at a storage structure which is sufficiently fine grained. The definition of what constitutes a sufficient granularity is a pragmatic decision on what is your desired level of cooperative work management functionality from a development methods point of view versus what are the acceptable cost in the cooperative work management software.

An object structure which is more fine grained, requires that we first determine the granularity which we consider appropriate. Choosing the appropriate granularity must be done for every sub-language. Each sub-language can be categorised as either hierarchical or flat. For those sub-languages that are categorised as hierarchical, we must determine which is the most fine grained language construct that is subject to hierarchical refinement. That language construct will then become the most fine grained development object. For those sub-languages that are categorised as flat, we must determine a suitable composition of language constructs to form a conceptual view. The conceptual view should be defined such that it can be used as a standalone object in the systems development process.
Generic procedure to derive specific object structure

The generic procedure to derive the specific object structure for a specific conceptual model (e.g., data flow diagrams, PrM diagrams, etc.) consist of the following three steps:

i Decide on granularity of the development objects.

ii Group facts into development objects as specified by the decision on granularity.

iii Add on attributes and structure as required by the cooperative work management functionality, e.g., version identifier, transaction identifier, owner identifier, etc.

7.4.3. The granularity of the PrM and PhM

Based on the discussions above, this section presents the definition of a feasible granularity of the PrM and PhM sub-languages.

If we use the decisions on granularity presented in this section, a feasible object structure for PPP may be derived from the fact structure for PPP given in appendix F. The PPP storage structure. An example of this is presented.

The PrM granularity

The components of PrM diagrams are processes, data stores, data flows, timers and external entities. The PrM diagrams comprise a natural hierarchy through the decomposition of processes. Revision of a process happens through a revision of its decomposition. The data stores and the data flows are further described (decomposed) by a selection of components in PhM diagrams. The remaining components, the timer and the external entity, are not refined through hierarchical decomposition. From a version handling perspective, refining a timer or an external entity is equivalent to deleting the old timer or external entity and inserting a new one. Thus, the "bottom level" decomposition of processes, data stores, and data flows of PrM diagrams are the lowest level development object that will exist in several versions.

The PhM granularity

The PhM diagrams are categorised as flat models. The feasibility of keeping versions of entities, attribute relations, objects, subset relations, connections, data types, and data type compositions is in our opinion questionable. The feasibility of keeping versions of "entire" diagrams is much more immediate. The size of the diagram would be entirely up to the systems developer. The conceptual view concept offers to structure the PhM models into diagrams of manageable size.

The example revisited

As an example, consider the component structure graph for Replenish Stock version 1.2, that is depicted in figure 5-13. This compo-
nent structure graph reflects the data flow diagrams depicted in figures 5-11, 5-11, and 5-12 respectively.

---

**Fig. 7-8: Object structure highlights**

Some of the PPP facts necessary to represent these three Data Flow diagrams are

- **prm_diagram**:
  - `prm_diagram(d1000, - , - , (p1001,p1002,p1003), (s1001,s1002,s1003), (a1001,a1002), - , - , (f1001,f1002,f1003,f1004,f1005,f1006,f1007,f1008))`
  - `prm_diagram(d1020, d1000, p1002, (p1021,p1022,p1023), - , - , (f1004,f1005), (f1006,f1007), - , (f1021,f1022,f1023,f1024,f1025,f1026,f1027))`

- **process**:
  - `process(p1001, p1010, (f1001,f1002,f1003), (f1004), 0, "Estimate future need")`
  - `process(p1002, p1020, (f1004,f1005), (f1006,f1007), 0, "Find best supplier")`
  - `process(p1003, - , (f1006,f1007), (f1008), 0, "Place order")`

- **store**:
  - `store(s1001, - , (f1003), 0, "Back orders")`
  - `store(s1002, - , (f1002), 0, "Order history")`
store(s1003, -, (f1005), 0, "Supplier data")
agent(a1001, -, (f1001), 0, "Stock clerk")
agent(a1002, (f1008), -, 0, "Supplier")
flow(f1001, a1001, p1001, 1, "Replenish part")
flow(f1002, s1002, p1001, 0, "Order history")
flow(f1003, s1001, p1001, 0, "Back orders")
flow(f1004, p1001, p1002, 3, "Order quantity")
flow(f1005, s1003, p1002, 0, "Supplier data")
flow(f1006, p1002, p1003, 0, "Chosen supplier")
flow(f1007, p1002, p1003, 3, "Order quantity")
flow(f1008, p1002, a1002, 2, "Order")

The complete set of PPP facts that is necessary to represent these three Data Flow diagrams may be found in appendix G.

According to the steps given in the procedure to derive a specific object structure that was presented in the previous section, the PPP facts in appendix G which represents the Data Flow diagrams in figures 5-4, 5-11, and 5-12, may be grouped to form a number of development objects. This is illustrated in figure 7-8.

The complete set of objects may be found in appendix H.

Observe that the objects that comprise the development object structure in figure 7-11 correspond to the objects in the component structure graph depicted in figure 5-13, with the added complexity arising from the decision to consider both the processes, the data stores, and the data flows to be the most fine grained objects of PrM.

7.4.4. Alternative storage solutions

The architecture of figure 7-6 utilises a DBMS to store the development objects of the PPP+CWM tool. Several alternative technical solutions exist to realise persistent storage of the development objects. In this section we discuss four different alternatives: Firstly, we discuss the alternative of using Unix files to store the development objects. Secondly, we discuss the alternative of using a relational database. Thirdly, an extended relational database alternative is discussed. Finally, a document storage solution based on ODA is discussed.

We have chosen not to include an object oriented DBMS alternative for several reasons: Firstly, the object oriented database technology is not considered mature. Secondly, the object oriented databases have not yet been commercially accepted. Thirdly, our experience and knowledge is much deeper with the four alternatives that have been chosen than with an object oriented DBMS alternative. Finally, our extended RDBMS alternative is believed to exhibit many of the features of object oriented databases that are desirable in this context.

File based alternative

A solution where the development objects of the chosen combined PPP+CWM tool architecture (depicted in figure 7-6) are stored in Unix
files will store the development objects as collections of Prolog facts "bundled" with information necessary to support the CWM functionality, in separate Unix files, one "bundle" per file. This alternative is illustrated in figure 7-9.

This is an attractive alternative for several reasons: It is conceptually simple, it is similar in approach to the present file based implementation, and it is simple to optimise for good performance.

On the other hand, the file based alternative also involves drawbacks: The programming interface for the CWM component is on a very low abstraction level, and all low level concurrency and consistency handling is left to the CWM, a set of tasks which otherwise is left to a DBMS. In fact, a part of this solution would be to implement a "primitive" DBMS.

![Fig. 7-9: A file based solution](image)

The current PPP implementation also uses Unix files to store the development objects [62]. Weiblen augmented this implementation by implementing a selection of the CWM functions presented in this thesis, e.g., object administration, bookkeeping, and mapping versions of objects to Unix files, according to a file based storage solution alternative [153].

**Relational database alternative**

In an RDBMS solution a relational schema represent the data necessary for CWM functionality, e.g., the structure of objects, the versioning information. In addition, the content of the development objects must be stored. There are two different ways to store the contents of the objects; either by developing a relational schema closely reflecting the meta model of PPP, i.e. representing hierarchical and flat objects as a series of relational tuples, or by storing the contents of objects as Unix files which are referred to from within the relations. The latter alternative is depicted in figure 7-10.

The advantages of this alternative are: It uses standard well proven technology, the database system will handle all low level concurrency and consistency keeping, and it offers a programming interface for the CWM component which is on a sufficiently high abstraction level.

The drawbacks of this alternative are: The representation of the contents of the development objects is not very elegant, and the "impedance mismatch" between a relational representation of data and a feasible data
structure to be used by the PPP analysis and design tool components is not desirable.

Currently, a new implementation of the PPP tool is being undertaken [100]. This implementation will implement the CWM functionality presented in this thesis with the RDBMS storage solution illustrated above. The implementation has chosen to map the PPP meta model directly into relational tuples, thus avoiding the mix between Unix files and an RDBMS.

Fig. 7-10: An RDBMS based solution with Unix files

Extended relational database alternative

During recent years many relational databases have been equipped with the sequence data type. This feature is now supported by major DBMS vendors, e.g., Oracle, Sybase and Ingres. The sequence is an addition to traditional data types like character and integer.

The sequence datatype is also frequently called BLOB (Binary Large OBject). This gives a more intuitive notion of its potential usability. BLOBs may be of arbitrary size. An attribute in a relation may be of type sequence. An attribute of the sequence type may be interpreted as a repeating group where the group consists of a predefined set of sequence fields. Each sequence field may be of any data type, except sequence and repeating group. I.e., a sequence may be regarded as a sub relation stored as an attribute in a relation. There is no practical upper limit on the number of elements in a sequence.

Fig. 7-11: An Extended RDBMS based solution
The properties of the *sequence* makes it very useful for storing arbitrarily large development objects. We may therefore use standard relations for the structure information and the versioning information in a PPP+CWM tool. The contents of the object may be stored as a *sequence*. This is illustrated in figure 7-11.

The advantages of this alternative are: It uses standard well proven technology, it has the full functionality of a database system, the programming interface for the CWM component is sufficiently high level, and the contents of the object is stored together with the information about the object in a uniform way.

The drawbacks of this alternative are: The "impedance mismatch" between the relational representation and the feasible data structures in the CWM component is not desirable.

During 1992 and 1993 a CWM front-end to PPP with the functionality that has been explained in this thesis, was implemented by Kimmel and Bjørnøstad in student projects supervised by the author [88][20][21]. The feasibility of the front-end implementation was further investigated by Bjørnøstad in his diploma thesis by implementing a multi user document processing system based on Emacs and Latex [21]. The implementation was based on the TechRa extended relational database system of KVATRO [91] and stores the contents of the documents as BLOBs.

The detailed database schema for this front-end implementation is presented in appendix I.

### PPP on ODA

The last alternative we consider for representing the development objects is to store them as ODA documents. There is a number of document systems on the market which use the document as the granularity in their storage solution, e.g., Lotus Notes [107]. We choose to take ODA as an example of these systems. A very brief description of ODA was given in section 7.4.2.

ODA is proposed as a standard for document representation. If ODA succeeds it means that manipulation and interchange of such documents is approaching universal transparency. CASE tool specifications modelled as ODA documents will nevertheless only be properly understood by software compatible with the specific CASE tool.

The (hierarchical) structure of the development objects is mapped to a corresponding ODA specific logical object structure. The content portions are filled with the corresponding facts from the development objects.

The advantages of this alternative are: Bearing in mind that a substantial part of the specifications of a system will be textual documents, we get a uniform representation of the complete specification. Furthermore, this representation is an international standard. Finally, the universal interchangability thus achieved is attractive although the documents may not be properly understood by software not compatible with the CASE tool that generated the document.

The single most important drawback of this alternative is: There are currently no commercially available database systems specially geared to-
wards ODA, and as far as we know there is no generally accepted proposal for how an ODA document should be stored using any of the accepted commercial database systems.

However, the principle outlined in this alternative may also be used with commercially available document storage systems like Lotus Notes from Lotus Corporation, DocuLive from Siemens Nixdorf, etc.

7.5. Chapter summary

This chapter has contributed to a validation of the proposed framework for cooperative work management by developing a repository architecture for the experimental ICASE environment PPP. A set of requirements to a repository solution for PPP has been developed. Alternative integration strategies between the PPP tool and the CWM functionality was discussed before we developed a proposal for an architecture which integrated the CWM with the PPP tool. Finally, the schema for the repository for PPP was developed. This included the development of a detailed example showing the resulting development object structure of a PPP model with the chosen CWM approach.

Tight integration of cooperative work management functionality with a CASE tool can only be achieved when having access to the meta model of the tool and having the possibility to augment it. Secondly, one must be able to modify the tool so as to integrate the specific CWM core functions and the CWM specific data access and user interface functions. Hence, this can only be done by a tool vendor and not by a tool user.
8. Conclusions and future work

This chapter contains the main conclusions of the work, the claimed contributions, and suggestions for future work.

8.1. Main conclusions

The main conclusions to be drawn from this work are the following:

- Support for team style of working in contemporary CASE technology is weak. It varies from nothing to a very restricted split of development objects where there is no overlap or sharing among workspaces.

- An improved approach to team work should be based on true sharing of development objects. In our proposal objects may be checked out to many transactions simultaneously. Synchronisation among transactions is achieved by publishing and subscribing to local versions of the objects. When competing transactions desire to update the repository, negotiations are forced. The negotiations are either synchronous or asynchronous, and result in a single version of the object to update the repository. Integration of groupware technology provides support to the negotiations.

- Many contemporary information system specification models are too complex to be understood without the use of suitable abstraction mechanisms. The abstraction mechanisms help reduce complexity and highlight selected properties in the specifications. The framework for cooperative work management have been applied to such complex models. Improvements to the framework have been suggested.

- An integration of the proposed framework with a database has been developed. The product of the integration comprises a repository for the experimental ICASE tool PPP. A prototype of this integration has been implemented, but has been restricted to textual objects.

- To achieve tight integration of cooperative work management functionality with a CASE tool, one must firstly have access to and the possibility to augment the meta model of the tool. Secondly, one must be able to modify the tool to integrate the specific CWM core functions and the CWM specific data access and user interfaced functions with the tool. Hence, this can only be done by a tool vendor.

8.2. Claimed contributions

This thesis has contributed to

- A more flexible approach to information systems development when done by teams. The approach permits overlap of development objects among transactions. Update conflicts are resolved through negotiations supported by groupware technology. The approach provides improved relationships between synchronous and asynchronous modes of working.
• Integration and utilisation of feasible software components from the
groupware field. The improved relationship between synchronous
and asynchronous modes of working is due to the integration of ver-
sion and configuration control, management of development trans-
actions, and negotiation support. The negotiation support comprises
groupware components such as joint editing functionality, video con-
ferencing, and multi media annotation and mail functionality.

• Augmenting the functionality of repositories for CASE tools with
functions that permit true sharing of development objects among a
team of developers. It has been shown that the approach may be in-
tegrated into already existing tools, provided the meta model imple-
mented by the tool is externalised.

8.3. Directions for future work

We propose the following items for the future

• Extending the proposed framework with items that were suggested
in section 6.5. Neither of these extensions require modifications to
the fundamental concepts in our proposal, they rather serve to im-
prove the usability of the approach.

• The current prototype of the CWM front-end is not integrated with
any software to assist in synchronous or asynchronous negotiations.
A next step would be to integrate the front-end with feasible group-
ware tools to achieve the specified negotiation support.

• The current prototype of the CWM front-end is not integrated with
PPP. A next step would be to integrate the front-end with PPP or an-
other feasible CASE tool.

• Finally, we would consider it important to conduct a field test of the
approach with the purpose of verifying the feasibility of the approach.
The field test would have to use the integrated prototype with both the
negotiation support and PPP integrated.
Appendices

A. IEF an example of an ICASE tool
B. IPSEs - Integrated Project Support Environments
C. PCTE - Portable Common Tool Environment
D. IBM's AD/Cycle and repository manager
E. A meta model for PPP
F. The PPP storage structure
G. PPP facts of a detailed example
H. Object structure of a detailed example
I. The repository schema for the ERDBMS alternative
A. IEF an example of an ICASE tool

IEF consists of two components, the workstation software and the mainframe software [142][141]. Figure A-1 depicts the IEF architecture.

The workstation software runs on single user workstations, mainly MS/Dos and OS/2, and consists of four toolsets: The planning toolset, the analysis toolset, the design toolset, and the construction toolset. The workstation toolsets consist of single user tools that operate on specifications that have been checked out from the central encyclopedia.

![IEF Architecture Diagram]

The mainframe software runs on a shared computer, mainly running MVS, and consists of the construction toolset and the central encyclopedia.

The IEF architecture is comprehensive and offers extensive life cycle support. There is tight integration between the different tools and toolsets. Tight integration is facilitated by a fine grained specification repository based on an integrated meta model covering all the tools and toolsets. The architecture offers:

- PC-based development combined with mainframe project coordination. The workstation based tools offer sets of diagramming tools for system design. All tools operate against the same integrated specification repository. OS/2 based generators and testing facilities offer a simulated production environment for application testing. The mainframe based central encyclopedia offers system integration and project coordination. Team members can work simultaneously on disjoint parts of the same model.

- A diagram based run-time testing tool which offers a diagrammatic view to a system to facilitate testing and tuning.

- Generation of operational 3rd generation code. The generated code is never modified by the systems developer. Maintenance is done by altering diagrams and regenerating appropriate blocks of code.
• Split of work within projects using the subsetting mechanism. Many developers are allowed to work on a project. Overlap between sub-models is not allowed. By the subsetting mechanism a model is divided into logical sub-models. IEF preserves the logical relationship of each developer's work to the rest of the model.

• Building of integrated systems. The Information Engineering method encourages corporate data modelling resulting in building integrated systems designed for data sharing.

• Environmental independence, so that it is possible to build systems for multiple platforms from a single design. Code can be regenerated for new platforms.

A.1. The single user toolsets

The workstation software runs on single user workstations, mainly MS/Dos and OS/2. The workstation toolsets consist of single user tools that operate on specifications that have been checked out from the central encyclopedia, while the mainframe toolsets are partly single user tools, and partly multi-user tools. A single user toolset comprises four toolsets: The planning toolset, the analysis toolset, the design toolset, and the construction toolset. The construction toolset runs on both the workstation and the mainframe.

The planning toolset

The planning toolset supports a strategic top-down analysis, Information Strategy Planning, of the business as a whole. The plan identifies high level requirements and produce three architectures: The information architecture, the business system architecture, and the technical architecture.

The information architecture consists of a model of business information, business activities, and their interaction. The business system architecture formulates potential business areas to support the information architecture. The technical architecture describes hardware and software environments to support the business system architecture.

The planning models are entirely conceptual, i.e. they are completely independent of physical or technical details of implementation. The models provide a framework for superimposing such details during later stages.

The analysis toolset

The analysis toolset supports business area analysis. A business area is a set of related data, activities, and their interaction. A business area analysis consists of a data model, an activity model, and an interaction model. These models are typically subsets of components created by the Planning Toolset during Information Strategy Planning.

The business area models are entirely conceptual. They represent the essence of the business area. The models document names, interactions, meanings, quantities, conditions, and logic. IEF follows rules for map-
ping conceptual objects to physical objects, thus the detailed rules of business area analysis trigger a number of automated decisions.

The design toolset

The design toolset supports two stages of the Information Engineering method: Business System Design and Technical Design. The design activities consider implementation details such as procedure flows, user screen formats, and database management systems.

The design activities set in motion a number of automatic transformations to the model. The results conceptually represent the physical implementation of a business system. Objects map more closely to end results than in previous stages. From these objects, code and database generation take place during the construction stage.

The planning, analysis, and design toolsets consist of a number of overlapping diagramming tools as depicted in figure A-2.

<table>
<thead>
<tr>
<th>Diagramming Tool</th>
<th>Planning Toolset</th>
<th>Analysis Toolset</th>
<th>Design Toolset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organisational Hierarchy</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix Processor</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Matrix Definition</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Data Modelling</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Activity Hierarchy</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Activity Dependency</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Action Diagram</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Structure Chart</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Action Block Usage</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Business System Definition</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Dialogue Flow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screen Design</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Prototyping</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Data Structure</td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Fig. A-2: Workstation tools and toolsets [141]

The construction toolset

The construction toolset may run either on a workstation or on a mainframe. The toolset allows for generation, testing, modification, and retesting of entire business systems or subsystems. Operationally complete COBOL or C programs, DBM or DB2 databases, interfaces to non-DB2
databases, 3270 or MFS user interface screens, and transaction control code (including JCL for batch applications) are generated.

The toolset prepares for automatic generation for five different target environments: OS/2 (IEF/Application Execution/DBM), MVS (TSO/DB2), IMS/DC/DB2, CICS/DB2, and Batch/DB2. Developers can selectively generate or regenerate screens, procedures, collections of procedures, or the whole business system. Evolutionary prototyping is supported. A generated system component may execute immediately under TSO or OS/2 regardless of its final target environment.

A.2. The central encyclopedia tools

The central encyclopedia tools run on a mainframe computer. The preferred operating system is MVS. The central encyclopedia is an object-oriented implementation of a DB2 relational database. The database is viewed through detail windows in the PC toolsets. Each object in the encyclopedia has its own detail window to describe it.

The mainframe tools provide coordinated access to the Central Encyclopedia. The tools enable centralised reporting and serve as a focal point for managing and distributing specifications among workstations.

Project teams require means of coordinating and synchronising team activities, and are supported by locking mechanisms and merging techniques. The tools that support team working include facilities for model distribution, model merge, and version control.

Model distribution

IEF distributes sub-models to workstations through a technique called subsetting. Subsetting guarantees integrity of the model by requiring that the different subsets are disjoint with respect to delete and modify authorisation on individual objects, and by keeping the workstation copies separate. This permits team members to share the same model without contention.

Subsets are defined using the central encyclopedia tools. A developer defines a subset by scoping it to include specific objects or types of objects. Objects have four different authorisation rights: Delete, modify, read-only, and access. Objects included in the subset with delete or modify authorisation are locked within the model. The lock is held by the encyclopedia until the owner of the subset returns the subset to the mainframe. During the interim period, other users may read the locked objects but can not update the locked objects in the subset. Locking is based on the user's intent to change an object rather than simply read it. For example, a user may request delete and modify authorisation to a procedure, its action diagram, and its associated screen, and read authorisation to related entity types.

Model merge

The combination of two models from the central encyclopedia to create a single composite model is called model merge. This allows separately
created models to share common data definitions, while processing logic from both models are preserved.

A model merge requires one of the models to be specified as dominant and the other as subordinate. The merging produces a composite model based on similar names and structures. Three merge options may be specified: Merge Trial, Merge New, and Merge Old. Merge Trial reports the results of the model merge without actually performing it. Merge New leaves the original models intact and creates a new composite model. Merge Old merges the subordinate model into the dominant model. The subordinate model is left unchanged and the dominant model is updated. The details of name or structure conflicts are reported in a Merge Conflict Report. When discrepancies occur, information from the dominant model prevails.

The difference between model subsetting and model merge is that while model subsetting creates subsets where no overlap is allowed, model merge is a way of synthesising work that has already overlapped.

**Version control**

IEF supports multiple copies of the same model at different stages of development, testing, and production. The multiple copies are called versions. Changes may be replicated and transferred between versions. For example, developers can transfer the changes from a development copy to test copies and production copies of the model. Different versions of a model that support different life cycle stages are said to belong to the same model family. The process of transferring objects between family members is known as migration. The objects selected for migration are called aggregate objects. Aggregate objects represent one or more related objects in a model. The granularity of the IEF version control allows aggregate objects to range from the very fine grained, e.g., a single attribute of an entity type, to the coarse grained, e.g., a selection of entity types and related procedures.

**Miscellaneous model management functions**

In addition to the functions for subsetting, model merge, and version control, IEF provides functions to copy, delete, rename, convert, and check models for consistency. Statistical and historical data may be reviewed. New models may be created from subset definitions.

**Model security**

Initially models are built locally on workstations, and then checked into the mainframe resident Central Encyclopedia (uploading). Authorised users can check out models or subsets within a common project. The checkout procedure downloads a copy of the model to a Local Encyclopedia and marks it as the active copy. To prevent model overlap, only one copy of a model or subset can be modified at a time. The Central Encyclopedia is updated with changes to models. The updates make the changes available to other team members and provide a backup of the work done at individual workstations. When ready to release an active copy, a developer checks in the copy by uploading it.
The Central Encyclopedia is protected from unauthorised access by enforcing three levels of security. Only designated project team members can modify or delete objects in models, based on the degree and scope of the approved access.

The highest level of access belongs to the encyclopedia administrator, who coordinates the overall use of the Central Encyclopedia. The encyclopedia administrator has access to all models. A second level of access is granted to project coordinators by the encyclopedia administrator. The project coordinators can check out and perform management functions on models and subsets assigned to them. The lowest level of access belongs to user analysts, who can check out specific models and subsets and generate user reports.

Reports

There are two kinds of predefined reports: Model user reports and model management reports.

The model user reports relate to encyclopedia content. They are accessible to all users, and include reports for: Entity definition, attribute definition, function definition, elementary process information view definition, scoping object where used, and expansion conflict.

The model management reports relate to encyclopedia administration, and are accessible to the encyclopedia administrator and project coordinators. They include reports for: User access, model access, duplicate object name, model contents, and model index.

Interfaces

IEF supports three levels of interface: Seamless, independent, and public.

The seamless interface allows Central Encyclopedia access to take place from within the IEF workstation toolsets. This requires that the workstations are equipped with IEF supported communication cards.

The independent interface is for workstations that are interfaced to the mainframe through any other means than through the IEF supported communication card. Accessing the Central Encyclopedia is done in two steps: (1) Preparation for file transfer using the IEF facilities on workstation or mainframe. (2) Logon to TSO and transfer using independent facilities. The information retained and exchanged between the Local and Central Encyclopedias is still the same.

The public interface provides the IEF with an open architecture through an import/export utility. It lets IEF users read project data from other tools and convert IEF data to a format that other tools can use. The import function creates an IEF compliant model from files created by other CASE tools or data management products. After a model import, a Public Interface Import Report describing the objects that have been imported into the model is automatically produced. The export function makes IEF models available to other CASE tools and user applications. This function delivers a set of encyclopedia tables together with a schema. The ability to do selective
retrieval from the Central Encyclopedia makes the export function useful for creating ad hoc reports via SQL or a report writer program.
B. IPSEs - Integrated Project Support Environments

An IPSE is a life cycle product that puts emphasis on integration. IPSEs support the development life cycle by integration of point tools for requirements analysis, via system design, through to system construction, testing, and maintenance. The integration of tools is normally realised within the framework of a project. The granularity of the design objects tend to be that of a document rather than a more fine grained approach. That is, the focus is on managing the process of software development and the design products created during that process.

General description

The IPSE has been defined as follows [2]:

"The common understanding of an IPSE is that it should contain a compatible set of specification, design, programming, building and testing tools supporting a development method that covers the entire life cycle, together with management control tools and procedures, all using a coordinated and consistent project database."

"What is more, given the certainty that no one particular programming language will emerge in the medium term accepted as a standard meeting all needs, such IPSEs will require multi-lingual capabilities."

A generic IPSE architecture consists of three main toolgroups for acquisition and development, project management and version & configuration control interface, and project dictionary.

The acquisition and development tools are integrated from potentially several independent vendors. These are tools for requirements analysis, software design, programming, and system maintenance. The various point tools use the project dictionary as their database. They access the dictionary through the project management and version & configuration control interface and thus provide versioned configurations of design objects within the chosen project management framework.

The project management and version & configuration control interface is an interface to the project dictionary and provides versioned configurations of design objects and project management data. The interface maintains design object configurations in conformance with the versioning and configuration definitions given, and with the access rights and project work structure defined. In this work it utilises basic tool service functions such as version control, automatic system building, and document preparation facilities.

The project dictionary is usually a standard database system. The point tools used for software development and maintenance put requirements on object granularity, object relationships, storage/retrieval functionality, etc., that points towards an object orientated approach rather than a relational approach for the database. The dictionary is
often split in a project database and a software design database. Whereas the traditional IPSEs have put emphasis on the former, the next generation IPSEs are expected to integrate the two, and thus support both project and software design concepts.

The SFINX definition of an IPSE

The Esprit project SFinx [129], defines an architecture for a generic IPSE (figure B-1). The architecture comprise 4 layers of tool services on top of the hardware platform: The software engineering environment kernel, the basic tool services, the software development process management, and the specific tools (for requirements capture and analysis, software design and construction, and maintenance).

![IPSE Architecture Diagram]

**Fig. B-1: IPSE architecture according to SFINX [129]**

**The SEE Kernel** provides high level operating systems like facilities such as process handling, I/O facilities, database management facilities, distribution support, and basic user interface support. The PCTE developed within the framework of the Esprit programme, is an example of a SEE kernel. Although it is a commercially available product, it has not yet reached wide acceptance in the market.

**The basic tool services** provide basic tool integration facilities such as version control, configuration management, automatic system building, document preparation, document management, basic data structure manipulation, security, and host-target communication.
facilities. The PACT developed within the framework of the Esprit programme, is an example of a basic tool services architecture.

The software development process management provide tools which act as a structured interface layer between the point tools and the dictionary. The structure in the interface layer is determined by the version and configuration structure coupled to the project management and change request management facilities.

The point tools provided with an IPSE can potentially come from several different vendors, and can be targeted towards requirements analysis, architectural design, detailed design, coding and unit testing, system integration, or maintenance.

B.1. Important IPSE issues

Important issues in the context of an IPSE are project management, version and configuration management, and data dictionary support.

Project management

The management of software projects share a number of features with project management in general. Software developers often use the general purpose project management packages. Standard packages for Pert and Gannt charts are also used. Project management of software projects nevertheless have certain peculiarities. The difficulties of estimating project time and resource requirements is the most prominent. An information systems project planning package should support estimation, planning, scheduling, and control.

Version and configuration management

Most design objects undergo a series of revisions, both during development and after delivery. There are two main reasons why modifications are made to a design object: (1) faults are discovered; the design process may not have resulted in objects with properties that are consistent with the specification for the design object. (2) requirements change; the world changes or the ambitions reflected in the requirements to the design objects changes.

Important functional areas for version & configuration management systems are versioning of objects, naming schemes for versions, constraints on updating, configuration management, and system rebuilding [23].

Versioning of development objects comprises the various physical instances of object versions. Versions may either be revisions or variants. Revisions reflect a chronological development, where each revision is developed to replace its predecessor. Variants reflect parallel development paths, where each variant is a sequence of versions.

Associated with the versioning is the problem of finding an efficient storage mechanism for object versions. Storing every version in its entirety is obviously a waste of space. There are efficient approaches that stores only the differences between versions, either through forward or backward deltas. With the forward delta storage technique,
the first version is stored in its entirety, and the next version is stored as the difference between the first and this version, and so on.

**Naming schemes for versions** ensure that all versions of development objects are given unique names for identification. The usual approach is to provide sequence numbers for the revisions and unique identifiers for the variants.

**Constraints on updating** deal with concurrent systems engineering aspects. The concurrency mechanisms prevents updating of frozen versions. Furthermore, updates may only be performed by the developer who has appropriate authorisation. Permission to update is normally only possible if the object is in the domain of the developer. Update permission may be granted and revoked by a mechanism of publishing and withdrawing among domains.

Design object versions will exist in different stages of development, from a mere sketch to the polished perfect. It is useful, particularly as a link to the configuration management system, to keep track of the status of each version of design objects. The particular status values to be used are project specific.

**Configuration management** relate different development object versions in a system to each other. Dependencies which result from all processes of system development are kept to reproduce given versions of objects, produce new components and versions from existing objects, and track dependencies between objects and repercussions of impending modifications.

**System rebuilding** takes care of regeneration of objects. Each development object has a set of objects required for its construction. The necessary information concerning the precise versions required to rebuild particular versions of objects and the full details of how the rebuild is done may be kept. In some cases the rebuild tasks may be automatically re-run, e.g., through compilation.

**Data dictionary support**

There are five characteristic properties that distinguish an IPSE from other life cycle products: (1) IPSEs integrate point tools from many sources, (2) IPSEs provide team support through sharing of data and communication between developers, (3) IPSEs provide tools to support all stages of the development life cycle, (4) IPSEs provide tools for project management, version control, and configuration management, and (5) IPSEs control access to both tools and data.

All of these characteristics point to an essential component in the IPSE architecture: the project database (dictionary, repository). The dictionary relates software development activities with tools, development organisation, and development objects (development deliverables). A meta model determines the degree to which the characteristic features of the IPSE may be realised. E.g., tool integration will only be possible if the meta model supported by the dictionary is compatible with the meta models of the point tools being integrated.

Consider the meta model used by the McDonnell Douglas Stradis method (figure B-2). This top-level model does not explicitly give the sub-classifica-
tion of the major entity classes (*Roles*, *Deliverables*, *Tools and techniques*, *Activities*, and *Steps*).

Data exchange between third party tools in contemporary IPSEs is difficult. The subject has attracted much research interest, and the next generation IPSEs is expected to have an improved solution to this problem. Today we see dictionaries that support adequate granularity for project data, but have a too coarse grained solution for the design data. It is expected that the next generation IPSEs will properly integrate the project and design data on an appropriate granularity level.

![Diagram](image)

**Fig. B-2: The McDonnell Douglas Stradis method meta model**

**B.2. Towards the information systems factory**

IPSE technology may be regarded as a merger between the Programming Environments and the Project Management Workbenches as depicted in figure B-3. The first generation IPSEs was the project support environment. The second generation IPSEs is the multi language, multi method, distributed environment. They provide databases to store all project information, and facilities to enable tools to use common user interfaces. Third generation IPSEs have been forecasted but have not yet substantiated. They are expected to rely on knowledge bases and to contain advanced AI based tools. The way onward from the third generation IPSE is called the *information systems factory*.

The terms *software factory* and *information systems factory* are closely related. A software factory differs from an IPSE in that not only does the former take into account the software engineering world, but also the organisation world [129]. While the software factory is restricted to production of software, the information systems factory adds facilities to support the development and maintenance of other information system components (hardware, data/information), and facilities to support the development and maintenance of complete information systems made up of such components [149][150].

The interpretation of the word *factory* in this context, is strikingly different from the accepted interpretation of the word [128] [149][150]. In a tradi-
tional factory products are manufactured from raw materials using existing designs. The production of information systems is however, more of a design problem than a replication problem. Thus the creative activities taking place in an information systems factory is analogous to the creation of the first production prototype of a traditional product, than to its mass production.

This means that the information systems factory designer must take into account constraints provided by elements from both the software engineering and the organisation worlds. (1) From the software engineering world there are software engineering environments, software development methods, and software development tools. (2) From the organisation world there are business environments for production, people’s organisation, people’s education, and communication practices.

![Diagram of tools and methods](image)

Fig. B-3: The evolutionary process of the information systems factory [129]

So the conceptual model in question does no longer reflect on only the software process and the psychology of those who engage in it, but rather on the whole range of activities undertaken by the software producing organisation.

Tully gives an appealing layering of information systems:

- Generic information systems factory which is a support system used to design and create specific information system factories by integrating components.
- Specific information systems factory which is a support system used to design and create information systems by integrating components.
- Information system which is a system that provides added value to a product or to an organisation process, by producing information in a useful form.

Oddy [114] suggests a four layered model of an information system development support facility:

- The information systems factory concept is a conceptual structure within which the development of information systems can take place.
• An information system development support facility, ISDSF, framework determines the physical architecture of a class of ISDSFs.

• An information system development support facility, ISDSF, virtual machine is composed of a number of components configured to run on a particular ISDSF framework.

• An information system development support facility, ISDSF, is an ISDSF virtual machine which has been sculpted to support the methods and procedures followed by the organisation that used it.
C. PCTE - Portable Common Tool Environment

When the Esprit programme was initiated the ambition to communicate not just results but software between various Esprit projects, was identified. It was highly desirable that projects could use software produced by other projects. For this to happen, the software produced across the various projects, would need to share a common base. The range of software produced by different Esprit projects were expected to span a diverse range of hardware, so the common base should be portable.

The original project for the Portable Common Tool Environment, PCTE, began in 1983 by the European Computer Machinery Association, ECMA. It was designed to be a distributed data model. In 1986, PCTE established its first programming interface based on the C language by Groupe Bull, Siemens, Nixdorf, GEC Marconi. In 1987, Ada interfaces were added to the model. In 1989, PCTE was extended to support an entity-relationship model and became known as PCTE+.

The architecture of PCTE supports an environment structured around a kernel which provides for all of a tool's requirements with respect to execution, inter-process communication, input-output, database access including concurrency control, distribution and user interface. The interface to the kernel provides a machine independent interface between the tools and the underlying system.

PCTE has three main components; (1) the object management system, (2) the program execution, communication, concurrency control, and distribution mechanism, and (3) the user interface facilities. We proceed to describe each of the components.

C.1. The object management system

The Object Management System (OMS) provides an object oriented database system by generalising the Unix file store concept.

An E-R model view

The OMS supports an entity-relationship model view to the structuring of data.

- Data is organised into objects (entities) connected by uni-directional links.

- All object types are always subtypes of another object type. There is a single root type object on which all other object types are based. Object types dir (for directory) and file are pre-defined.

- Object properties are inherited through the type hierarchy. The root type object has a set of attributes including owner, creation date, date of last modification, etc., which all objects in the database inherit. Specialisation of object types is through the addition of further properties.
Object types and link types may have attributes. Examples of scalar attribute types are boolean, integer, string, and date. There are no composite attribute types. The concept of null value for attributes does not exist. Thus attributes always have an initial value either supplied by the user or as a system default.

A relationship is modelled, and always maintained, as a bi-directional pair of links.

A link has one or more destination types, which indicate the legal object types that may be the destination of the link. The stability of the link determines whether the destination object of the link can be modified. The cardinality of the link determines whether it is scalar or multi valued.

A link has a category which can have one of three possible values: (1) The category composition dictates existence of the object pointed at. (2) The category reference indicates that the link is a user maintained link. (3) The category implicit indicates that the link is system maintained.

Schema definition sets

Object, link, and attribute definitions are organised into Schema Definition Sets (SDF):

- The attributes and relationships of object types are declared to the system through the PCTE Data Definition Language, DDL. An SDF may define some new object types, and add attributes and relationships to those new types and to existing types.

- SDFs may be added, modified, or deleted at any time provided they are not currently in use. The system does not need to be stopped.

- The schema is the aggregation of all the types, attributes, and relationships declared in the SDFs. An SDF defines aspects of the schema relevant to some tool or toolset, and may be viewed as the means by which a tool is integrated into PCTE.

- One or more SDFs make up a Working Schema. The working schema is the partial view of the database in use by a specific tool when it is executing. Only the current working schema is visible. Thus the data which a tool may access is limited to the types, attributes, and relationships defined in its working schema. Working schemas grow through the dynamic adding of SDFs as a tool executes and calls further tools.

Creating and maintaining the object base

PCTE offers functions for creating and maintaining the object base.

- There are functions for creating and deleting objects and links, and reading and maintaining the values of the attributes and links of objects, and attributes of links.

- There are functions for accessing the contents of an object, which treat the content as a file.
• There are functions for creating, deleting, reading, and maintaining SDSs.

• Objects do not have names, but are located by specifying a path to the object. Path names start from a reference object, naming links between objects using Unix like path names. Reference objects are either some well known object, or a previously accessed object whose location has been remembered.

• The object base is also accessible through a programming interface. There is no query language.

The database facilities are not intended as a means of modelling all of the data for some tool or toolset.

C.2. The process model

The basic mechanisms of PCTE are concerned with functions to manipulate the concepts of program and process: Program Execution, Communication Concurrency Control, and Distribution Mechanisms.

Highlights of the basic mechanism:

• A program is a set of instructions that may be executed by a process. A process is an instance of the execution of some program. A process is brought into existence by some parent process invoking execution of some program. A process executes within a dynamic context.

• Processes may send and receive messages. Messages are sent to and read from message queues, which act as mail boxes. The message queues are normally read in a first-come-first-served manner, but there are functions for developing priority queueing mechanisms.

• Concurrency control is based upon the locking of resources in the context of an activity.

• A resource is the basic item handled by the concurrency mechanism, and may be an object with its attributes and content, or a link and its attributes.

• An activity is the framework within which a set of operations on a resource takes place. Activities may be of class unprotected, protected, or transaction. Every process is initiated in an activity, ensuring that a context exists in which to lock resources.

• A process may start and terminate further activities. Thus a nesting of processes within activities and activities within processes is provided for.

• PCTE is a community of closely cooperating workstations communicating via an Ethernet LAN.

• It is transparent. The physical location of an object need not be known. The user could log on to any workstation in the network and access the object.

• It is a single distributed system.
• It is robust and the system degrades gracefully. Access to certain objects in spite of partial or complete loss of the LAN, or loss of one or more workstations, is ensured by replication.

• Processes can be started on both the local machine and on other machines in the system. If a program will only run on a particular workstation, it can be made to start automatically regardless of where it is invoked from.

C.3. The user interface facilities

The user interface facilities of PCTE is a genuine attempt to abstract the end-user interaction style from the application interface definition, and thus force tool developers to think in a device independent manner.

Highlights of the user interface facilities:

• The basic model is the WIMP (Windows, Icons, Menus, and Pointers) interface, which in itself assumes a bitmap screen and a pointer having at least three buttons with up/down transition notifications.

• It is useful to consider the interface as having two components: the Application Agent which provides the functions called directly by applications, and the User Agent which interacts with the end-user device and multi-threads the requirements of the application agents.

• Applications may have several windows which can be dynamically created and destroyed. Windows may be grouped, allowing for single tools to group and present several windows so that they look like a single window.

• To separate the tools view of output data from the end-user view, the frame concept is provided. Frames are rectangular, not constrained by the size of the display, and come in three types: Bitmap frames providing pixel level access, Text frames providing single or multi-font storage areas, and Graphics frames providing more complex shapes such as ellipses, arcs, polygons, and markers.

• PCTE provides menus, which may be: Static or pop-up, single or multiple choice, or simple or complex.

• Icons are graphic figures which are place holders for windows which have been "iconised" to save screen space. Icons are tool definable and dynamic update is allowed.

• All end-user actions are first seen by the user agent, which may interpret the action as the user agent’s responsibility, or transform the simple action into a request for a selection to be established. These end-user actions are passed with or without interpretation to the appropriate application agent.

• End-users may make selections on the visible information. A selection may be a graphic element or text. Selections may be adjusted. Operations such as delete, copy, or move on these selections can then be performed.
C.4. PCTE as an open repository

Recent development has "renamed" PCTE to be called the Open Repository. In a Reference Model endorsed by ECMA and submitted to ISO for standardisation [42] PCTE has been placed as the hub of a flexible repository. The reference model is illustrated in figure C-1.

At the bottom level are the operating system and network services. At the physical level, repositories may be implemented either directly over an operating system or on top of an existing (relational) database. Both approaches have advantages and disadvantages; implementing over a relational database makes porting easier whilst a direct implementation over an operating system (typically Unix) should have performance advantages.

Fig. C-1: Open Repository Reference Model

The repository itself provides an Entity Relationship Attribute (ERA) data model. This supports the most common data structures in system engineering: hierarchies, sequences, and networks of entities. The most common operations support navigating from entity to entity and updating attributes, relationships and particularly the content of single entities. ERA data models are natural for describing networks, and implementations may give good performance for the most common operations.

Compared to the rationale behind the project the PCTE initiative has only had limited success. Although PCTE has been used as the basis storage mechanism in several ESPRIT projects, and has been used in the Open Repository Reference Model by ECMA, it has not managed to establish itself as the prime choice for the majority of software building initiatives.
D. IBM's AD/Cycle and repository manager

In 1987 IBM launched their Systems Application Architecture (SAA). The SAA was meant to reduce hardware platform dependency both for application development and operation, and to support a uniform user interface to the applications.

D.1. AD/Cycle architecture

The AD/Cycle is the application development solution for SAA. It supports the SAA ideas in the following ways:

- Both IBM- and non-IBM-supplied workstation based tools will conform to the CUA (Common User Access) guide-lines.
- The AD/Cycle tools will support the development of applications written in the following SAA Common Programming Interface languages: C, Cobol, Fortran, PL/1, Report Program Generator (RPG), and procedures language.

Fig. D-1: AD/Cycle framework [72]

- Applications developed within the AD/Cycle framework can be targeted for execution in any of the SAA environments.
The programming interface to the repository will be a new element of the SAA Common Programming Interface (CPI).

The AD/Cycle framework was intended to support a comprehensive set of tools for application development and maintenance. The tools are conceptually grouped into sets that focus on specific life cycle activities. They will evolve to present a consistent user interface and share application development information over the entire life cycle.

The life cycle tools include the following:

- Modelling tools that support the definition of an enterprise business model and the validation of that model through prototyping.
- Analysis and design tools that refine business requirements described in the enterprise model into application and database design information.
- Languages, application generators, and knowledge based products for code development.
- Testing and maintenance tools for regression testing and test coverage measurement and analysis.

The IBM AD/Cycle framework is depicted in figure D-1. The AD/Cycle is a set of offerings for the support of application development throughout the development life cycle.

D.2. Cross life cycle activities

Cross life cycle activities extend across many or all of the life cycle phases. The activities include process management, project management, impact analysis, documentation, and reuse as depicted in figure D-2.

![Diagram of cross life cycle activities]

Fig. D-2: Cross life cycle activities [72]
• Process management may be done in two ways; user initiated and process model initiated.
  - The user initiated process management allows you to assign work items (application components, design products) to developers. The sequence of activities is left to the developers.
  - The process model initiated process management allows the definition of a process model describing application components, activities related to these components, and the sequence for performing these activities. A method to guide and control the developers work may thus be provided.

• Project management information include people assigned to a development project, work items for which they are responsible, overall shedule for the project, current status of the project shedule, projected cost of the project, and current cost of the project.

• Impact analysis evaluates the effects of specific changes to components of your application systems before these changes occur. This analysis is based on the application development component information, including relationships between components, that are maintained in the repository.

• Documentation includes the production of documentation using existing application development information.

• Reuse of application components will be promoted by maintaining an inventory of reusable application components, using information maintained in the repository. Developers are urged to identify likely candidates for reuse during early design activities, and structure these with reuse in mind.

• IBM encourage non-IBM vendors to support functions utilising this information.

D.3. AD/Cycle platform architecture

The general AD/Cycle platform architecture is depicted in figure D-3. The AD/Cycle platform comprises:

• CUA, Common User Access, Guide-lines represent a set of guidelines for tools and other SAA functions that provide a consistent appearance and interaction characteristics at the screen to application developers. This means that all users will access and use the work-station based tools supported by the AD/Cycle, across the life cycle, through the same graphic oriented interface supported by OS/2 presentation services.

• PWS, Personal WorkStation, Services represent the set of common services usable by PWS resident tools, eg., Presentation Management, File Services, Communication Services.

• Work Management help the user to sequence tool invocations. Three forms of work management are supported; direct invocation, invocation through object selection, and invocation controlled by a process mechanism. The process mechanism isis tailorable at installation.
- Tool Services Interface is a set of services that provide a high level interface to the data used across the life cycle by AD/Cycle Tools.

- Repository CPI, Common Programming Interface, is a set of guidelines that provides standard access to the repository.

- Repository Services is a set of Entity-Relationship and Object services that enable centralised management and control of all components of AD/Cycle.

- Information Model is an IBM supplied conceptual model of the data used across the life cycle, that is extensible by tool vendors and customers. The information model has two layers; the Object level and the ER level.

  - The Object Level comprises a set of objects that define a high level, abstracted representation.
  
  - The ER Level comprises a set of entities and relationships that define a discrete, low level representation.

Fig. D-3: AD/Cycle platform architecture [72]

D.4. The repository manager

The IBM definition of a Repository reads:

“A Repository is a database of specifications”

“A specification is a statement of particulars, especially one describing materials, dimensions, and workmanship for the building, installation, or manufacture of something.”

Highlights of the repository manager:

- Entity Relationship Model

- Three Views of Data; (1) the conceptual view which offers one single, global, enterprise wide definition of shared data, (2) the logical view which offers one logical view of data local to a function, filtered from the conceptual view per Repository manager function, (3) the storage
view which offers an implementation specific view of data (mapping to DB2 tables).

- Policies are coded in REXX and enforced at runtime by the Repository manager. There are four types of policies; (1) security policies which grant or deny access to repository data and tool functions, (2) integrity policies which enforce attribute constraints, and add/delete relationship type constraints, (3) derivation policies which automatic calculation of values on an event condition, and (4) trigger policies which call a function on an event condition.

- The IBM Information Model has four objectives; (1) to describe all application development data, (2) to describe the whole life cycle, (3) to be an open, published, and extendible architecture, and (4) to consist of two layers, i.e. the Object Model and the ER Model.

- Object Management

D.5. The information model

The information model consists of two layers; the Object level and the ER level.

The object Model level provides a set of objects that define a high level, abstracted representation of the information in the repository.

The object model schema is depicted in figure D-4.

Objects can be divided into two parts:

- Object data: Aggregation of some part of the ER level information model, and

- Methods: Reusable functions that can be performed against the object data.

A set of objects that uses the same methods is called an object type.

![Object Model Diagram](image)

Fig. D-4: The object model schema [45]

The ER level consists of the Enterprise model, the Design model, and the Technology model, figure D-5.
The Enterprise Model constitutes a conceptual representation of the enterprise and its constructs, and is a source for early previewing or prototyping.

The enterprise modelling involves analysing and documenting three facets of the business; (1) the processes that the business performs, (2) the information that these processes use, and (3) the business rules, policies, and practices that control these processes.

A simplified enterprise model is depicted in figure D-6.

Fig. D-6: A simplified enterprise model [72]
The Design Model constitutes the design level of the application, and acts as a source for prototyping or automatic generation.

The Technology Model is product specific, is directly populated by 3GL tools, and is the result of generation or transformation from the design model.

When IBM launched AD/Cycle they tried to provide an open framework for application development which would leave the developer freedom of choice with respect to development tools, but at the same time tie the development tools to IBM's operating systems and hardware platforms.

The AD/Cycle Information Model is already supported by a few CASE vendors, and a large number of vendors have promised AD/Cycle compliance [51]. However, acceptance and success with the development organisations is limited. It may be that the many delayed releases and the relatively weak tools (compared to the leading ICASE tools) have been a hindrance to success.
E. A meta model for PPP

As described in section 7.1, PPP consists of static modelling constructs, PhM, dynamic modelling constructs, PrM, constructs to describe low-level algorithms, PLD, and constructs to describe low-level definition of screens, UID.

The static modelling constructs facilitate a description of the static part of the universe of discourse, i.e., the entities, the way they are related and characterised. The dynamic modelling constructs facilitate a description of the dynamic part of the UoD, i.e., the processes (/activities), their internal structure, and how they communicate (the way they interact, what their means of communication is, and what they communicate). The link between the static and the dynamic sub-models is constituted by a modelling construct called item. Items are the “things” in the structural model that are manipulated in the dynamic model.

The description of the PPP meta model in this appendix follows the original definition by Opdahl [103]. We have developed a phenomenon model of all the PPP static and dynamic modelling constructs, and the way they relate to each other (figure E-1).

E.1. Static modelling constructs, PhM

The static modelling constructs facilitate a description of the static part of the UoD, i.e., the entities, the way they are related and characterised.

The static modelling constructs of PPP are: entities, attribute relations, objects, subset relations, connections, data types, and data type compositions.

We proceed to describe each of the static constructs separately.

Entities represent real world objects of interest in a specific modelling context. The entityclasses in a model will be partitioned into two subsets; objectclasses and connectionclasses.

An entityclass is a tuple (E) such that E is the entityclass. The entityclass E, must satisfy:

* E must be either an objectclass, or a connectionclass.

An entity instance is a tuple (e,E) such that e is the entity instance, and E is the entityclass. The entity instance e, must satisfy:

* E must be an entityclass and (e,E) an entity instance tuple, or E must be a connectionclass and (e,-,E) a connection instance tuple.

Objects are entities having an autonomous existence. Objects are either information objects or physical objects. They are diagrammatically depicted by rectangles.

An objectclass is a tuple (O) such that O is the objectclass.

An objectclass instance is a tuple (o,0) such that o is the object instance, and O is an objectclass.
Subset relations support modelling of objectclass hierarchies. There are three kinds of subset relations: subsets, disjoint subsets, and partitions.

A subset relation class is a tuple (SR,O,Rel,Os) such that SR is the subset relation class, O is an objectclass - called the owner class of the subset relation, Rel is either ss (subset), ds (disjoint subset), or dsp (partition) - called the subset relation type, and Os is a set of objectclasses - called the subset relation member. The subset relation class SR, must satisfy:
* There must be no objectclass O_i, O_i member of Os, such that O is, directly or indirectly, subset related to O_i.

* If Rel is either ds or dsp, there is no object instance oi, of the class O, such that there are subset relation instances (sr,oi,o_o,SR) and (sr_o,o_o,SR) where o and o_o are distinct instances of objectclasses O, O member of Os, and O_o, O_o member of Os, respectively. (No object instance owns several subset relation instances of the class.)

* If Rel is dsp, for all instances oi of the class O, there is a subset relation instance (sr,oi,o_o,SR) such that o is an instance of an objectclass O, O member of Os. (Every object instance is a subset relation instance owner.)

A subset relation instance is a tuple (sr,o1,o2,SR) such that sr is the subset relation instance, o1 is an object instance - called the owner instance of the subset relation, o2 is another objectclass - called a subset relation member instance, and SR is a subset relation class.

**Connections** represent entities whose existence depends on other entities. Connections are characterised by three aspects, i.e., cardinalities, coverage, and direction. They are diagrammatically depicted by an "upside-down" triangle circumscribed by a rectangle.

A connectionclass is a tuple (C,EDs) such that C is the connectionclass and EDs is a set, the entityclass description set, of tuples (E,Cov,Card,Dir) such that E is an entityclass, Cov, the coverage of the entityclass, is either f or p, Card, the cardinality of the entityclass, is a tuple (Low,High), Dir, the direction of the entityclass, is either f,b, or t. The connectionclass must satisfy:

* Low must be less than High in Card.

* There must be no indirect connection relation from C to E.

* If Cov is f, then for all entity instances (e,E), there is at least one connection instance (e,eds,C) where e is a member of ed.

A connection instance is a tuple (c,eds,C) such that c is the connection instance, ed is a set, the entity instance description set, of entity instances e, and C is a connectionclass.

**Attribute relations** describe the relevant aspects of entityclasses using data types. The attribute relations are characterised by Type, which is either identifier, attribute, repeating group, or quality, and Applicability, which is either mandatory, optional, or non-applicable. They are diagrammatically depicted by an arrow from an entityclass to a data type class, labelled to signify the appropriate characterisation.

An attribute relation class is a tuple (AR,E,D,Type,AppI) such that AR is the attribute relation class, E is an entityclass, D is a data type class, Type, the attribute relation type, is either id, attr, rg, or qual, and AppI, the applicability, is either mand, opt, or non. The attribute relation class AR, must satisfy:

* If Type is id then for all pairs of attribute relation instances (ar1,e1,d1,AR) and (ar2,e2,d2,AR), if d1=d2 then e1=e2.

* If Type is either id or attr then for all pairs of attribute relation instances (ar1,e1,d1,AR) and (ar2,e2,d2,AR), if e1=e2 then d1≠d2.
* If Type is qual then for all attribute relation instances (ar1,E1,d1,AR), E1 is always an entityclass.

* If Type is mand then there is no entity instance e, in E, for which there is not an attribute relation instance (ar,e,d,AR).

* If Type is non then there is no entity instance e, in E, for which there is an attribute relation instance (ar,e,d,AR).

An attribute relation instance is a tuple (ar,e,d,AR) such that ar is the attribute relation instance, e is an entity instance or - for instances of attribute relation classes whose Type is qual - an entityclass, d is a data type instance, and AR is an attribute relation class.

**Data types** represents the aspects of an entityclass which are relevant in the modelling context. Data type classes are linked to entityclasses by attribute relations. They are diagrammatically depicted as circles.

A data type class is a tuple (D) such that D is the data type class.

A data type instance is a tuple (d,D) such that d is the data type instance, and D is a data type class.

**Data type compositions** allow data types to be defined as Cartesian products of component types. Data type compositions are diagrammatically depicted as ovals.

A data type composition class is a tuple (DC,Ds) such that DC is the data type composition class, and Ds is a set of data type classes.

A data type composition instance is a tuple (dc,ds,DC) such that dc is the data type composition instance, ds is a set of data type instances, and DC is a data type composition class.

There are two additional static constructs we choose to introduce before we describe the dynamic constructs. They are items, and item sets.

**Items** are the objects stored in stores and resources or carried by flows. They constitute the link between the static and the dynamic models. Items are considered to be of interest in a modelling context either due to its physical existence, due to its information content, or both. Items are related to the static model such that the physical existence aspect or the information content aspect, or both, is represented in the static model as an entity.

**Item sets** are used to describe stores and resources. A store or a resource is characterised by the set of items, the item set, that it holds.

### E.2. Dynamic modelling constructs, PrM

The dynamic modelling constructs facilitate a description of the dynamic part of the UoD, i.e., the processes, their internal structure, the way they interact, what their means of communication is, what they communicate, and how they communicate.

The dynamic modelling constructs of PPP are: stores, resources, flows, flow terminals, processes, port trees, buffers, activities, superiors, and timers.
In the dynamic modelling part of PPP certain modelling constructs have been identified as being "nice to have", auxiliary constructs. The auxiliary constructs are: input/output relations, input/output dependencies, used items, process decompositions, input porttrees, output porttrees, inner porttrees, outer porttrees, input buffers, output buffers, inner buffers, outer buffers, events, and time parameters.

We proceed to describe each of the dynamic and auxiliary constructs separately.

The dynamic constructs of PPP are:

**Stores** are used to represent information and physical repositories. They are diagrammatically depicted by a rectangle with the right end missing.

**Resources** represent objects that are needed by processes for them to run. Resources hold "resource items", items that have certain restrictions placed on them. They are diagrammatically depicted by a circle with the resource name written inside.

**Flows** are modelling entities that relate stores, resources, and timers to processes, and processes to other processes. Flows represent item carriers. They are diagrammatically depicted by arrows, the arrow head indicating the direction of the flow.

**Flow terminals** are the modelling constructs sending and receiving flows. A flow terminal is either a store, a resource, a buffer, a timer, or nothing.

**Processes** are the modelling constructs producing and consuming items, updating stores and resources, and initiating and terminating flow uses. They are diagrammatically depicted by round-cornered boxes with a horizontal bar below its top edge.

**Port trees** are the structural counterpart to the visually oriented process ports. There is a port tree corresponding to any input and output port composition. The nodes of a port tree represent flow groups, leaf nodes correspond to buffers, internal nodes correspond to ports.

**Buffers** are constructs relating flows to processes. Buffers have no name, and are not shown in the diagram. They are simply assumed to exist where a flow is connected to a process.

**Activities** are sets of processes. They are a means of blocking processes for periods when they should not run. They are diagrammatically depicted by a rectangle with the activity name written inside.

**Superiors** are either processes or activities. Every process must have a superior. If a process has a decomposition, it is the superior of all the processes in the decomposition. If the process does not have a superior, it must be in the process set of an activity, in which case the activity is its superior.

**Timers** are either clocks or delays. Clocks are used to model events that are to occur at specific moment in time. Delays are used to model events that are delayed a certain time interval.

The auxiliary constructs of PPP are:

**Input/output relations** represent the item value part of a transformation. They are relations from sets of used input items of a process use and to
sets of used output items. Input/output relations are present in both input/output conditions and decision tables.

**Input/output dependencies** describe the temporal properties of an input/output transformation. They impose restrictions on when items may be consumed and produced during a process use. Input/output dependencies are present in both input/output conditions and decision tables.

**Used items** are needed in connection with input/output relations. A set of used items are either the set of items consumed by a process during process use, or the set of items produced by a process during a process use.

**Process decompositions** are sets of processes. The processes in a decomposition must be connected by flows, which are said to implicitly belong to the decomposition. The decomposition must have constructive properties to be valid.

**Input porttrees** define the relation between input flow groups. An input flow group is either an input port, or an input buffer. The logical relation defined by the input port may be either AND, OR, or XOR. Input ports have a type which is either singular, repeating, conditional, repeating conditional, or conditional repeating. Input ports may be triggering or non-triggering.

**Output porttrees** define the relation between output flow groups. An output flow group is either an output port, or an output buffer. The logical relation defined by the output port may be either AND, OR, or XOR. Output ports have a type which is either singular, repeating, conditional, repeating conditional, or conditional repeating. Output ports may be triggering or non-triggering.

**Inner porttrees** are ports “immediately” inside another port. A port containing no other ports, is called an innermost port.

**Outer porttrees** are outer to another port, if the latter is inner to the former. A port with no outer ports, is called an outermost port.

**Input buffers** stores an item received from its corresponding input flow, until the item is consumed for use by the process. Each process has a set of triggering input buffers.

**Output buffers** stores an item produced by the process, until the item is sent on its corresponding output flow. Each process has a set of terminating output buffers.

**Inner buffers** are buffers “immediately” inside another buffer. A buffer containing no other buffers, is called an innermost buffer.

**Outer buffers** are outer to another buffer, if the latter is inner to the former. A buffer with no outer buffers, is called an outermost buffer.

**Events** are constructs relating a dynamic modelling construct to its use. A sequence of events, called a trace, uniquely describes the model use. Events are either produce events, send events, receive events, or consume events.

**Time parameters** are (mean, deviation)-pairs which can be associated with all dynamic modelling constructs. The time parameters relate the modelling approach to the full time perspective, and are particularly useful when defining simulation models.
F. The PPP storage structure

The current PPP tool stores the specifications in Prolog fact files. The fact structure is divided into facts for overall system structure, facts for PhM diagrams, and facts for PrM diagrams. The meta model for PPP presented in appendix E reflects the specification of the PhM and PrM sub-languages [103]. The fact structure presented in this appendix is the one that is implemented in the current version of the tool, and as such deviates slightly from the original definition of the sub-languages.

The fact structure is reproduced in this thesis to be able to give a detailed example of how the proposed object structure may be implemented in PPP.

F.1. Fact for overall system structure

\texttt{infor\_sys(InforSysName, ApplicationList, ScenarioList)}

\textit{InforSysName} is the name of the information system.

\textit{ApplicationList} contains the identifiers of top level PrM diagrams.

\textit{ScenarioList} contains the identifiers of PhM diagrams.

F.2. Facts for PhM diagrams

\texttt{pm\_scenario(SceneId, EntityclassList, SubclassList, RelationshipList, AttributeList, SubclassList, OtherFigureList, LinkList)}

\textit{ScenarioId} is the identifier of the scenario.

\textit{EntityclassList} is a list of the entity classes in a scenario.

\textit{SubclassList} is a list of the subclasses in a scenario.

\textit{RelationshipList} is a list of the relationships in a scenario.

\textit{AttributeList} is a list of the attributes in a scenario.

\textit{SubclassList} is a list of the identifiers of groups of sub-entity-classes.

\textit{OtherFigureList} is a list containing the objects that are not among the basic components is a PhM scenario, but considered useful to enhance the diagram.

\textit{LinkList} is a list of the links in a scenario.

\texttt{entityclass(EntityId, FromList, ToList, Class, Des)}

\textit{EntityId} is the identifier of the entity class.

\textit{FromList} contains the identifiers of the links coming into the entity class.

\textit{ToList} contains the identifiers of the links going out of the entity class.

\textit{Class} is used to specify the type of the entity class.

\textit{Des} is a textual description.

\texttt{subclass(SubId, FromList, ToList, Class, Des)}

\textit{SubId} is the identifier of the sub-entity class.
**FromList** contains the identifiers of the links coming into the sub-entityclass.
**ToList** contains the identifiers of the links going out of the sub-entityclass.
**Class** is used to specify the type of the sub-entityclass.
**Des** is a textual description.

**subgroup**(SubGroId, FromList, ToList, Class, Des)
- **SubGroId** is the identifier of the sub-entityclass group.
- **FromList** contains the identifiers of the links coming into the sub-entityclass group.
- **ToList** contains the identifiers of the links going out of the sub-entityclass group.
- **Class** is used to specify the type of the sub-entityclass group.
- **Des** is a textual description.

**relationship**(RelId, FromList, ToList, Class, Des)
- **RelId** is the identifier of the relationshipclass.
- **FromList** contains the identifiers of the links coming into the relationshipclass.
- **ToList** contains the identifiers of the links going out of the relationshipclass.
- **Class** is used to specify the type of the relationshipclass.
- **Des** is a textual description.

**attribute**(AttId, FromList, ToList, Class, Des)
- **AttId** is the identifier of the attribute.
- **FromList** contains the identifiers of the links coming into the attribute.
- **ToList** contains the identifiers of the links going out of the attribute.
- **Class** is used to specify the type of the attribute.
- **Des** is a textual description.

**link**(LinkId, From, To, Class, Des)
- **LinkId** is the identifier of the link.
- **From** is the identifier of the first component linked by the link.
- **To** is the identifier of the second component linked by the link.
- **Class** is the type of the link:
  - 101: "sub"
  - 201: "id attribute"
  - 202: "attribute"
  - 203: "repeating attribute"
  - 204: "quality attribute"
  - 301: "in the relationship with full-1 type"
  - 302: "in the relationship with full-n type"
  - 303: "in the relationship with partial-1 type"
  - 304: "in the relationship with partial-n type"
- 401: “has a subclass group”
- 402: “has a disjoint subclass group”
- 403: “has a composing subclass group”
- 404: “has a distinct and composing subclass group” - partition
- 501: “is a member of a subclass group”
- 601: “is a component of a composite data type”
- 602: “is a repeating component of a composite data type”

Des is a textual description, indicating the type of the link.

**link**\_**fig**((LinkId, Dots, FirstDot, LastDot))

- **LinkId** is the identifier of the link.
- **Dots** is a list of all identifiers of the dots that link the lines of a link.
- **FirstDot** is the identifier of the dot which starts the link.
- **LastDot** is the identifier of the dot which ends the link.

**F.3. Facts for PrM diagrams**

**prm**\_**diagram**((DiagramId, HighDiagramId, ProcessId, ProcessList, StoreList, AgentList, TimerList, ExternalInputFlowList, ExternalOutputFlowList, OtherFiguresList, FlowList))

- **DiagramId** is the identifier of the diagram.
- **HighDiagramId** is the identifier of the diagram above the current diagram, i.e. the origin of the solved by relation pointing to this diagram.
- **ProcessId** is the identifier of the process for which the diagram is defined.
- **ProcessList** is a list of identifiers of the processes in the diagram.
- **StoreList** is a list of identifiers of the stores in the diagram.
- **AgentList** is a list of identifiers of the agents in the diagram.
- **TimerList** is a list of identifiers of the timers in the diagram.
- **ExternalInputFlowList** is a list of the identifier of the flows going into the diagram.
- **ExternalOutputFlowList** is a list of the identifier of the flows going out of the diagram.
- **OtherFiguresList** is a list of the objects that are not among the basic components is a PrM, but considered useful to enhance the diagram.
- **FlowList** is a list of the identifier of all the flows in the diagram.

**dia**\_**in**\_**port**\_**node**((DiagramId, PortId, InOutFlag, Level, Type1, Type2, ParentId, DeList))

- **DiagramId** is the identifier of the diagram.
- **PortId** is the identifier of the input port.
- **InOutFlag** shows whether the node is input port node or output port node.
- **Level** defines the level of this port within a composite port.
- **Type1** is AND, OR, or XOR.
Type2 is SINGLE, REPEAT, CONDITIONAL, R_C, or C_R.
ParentId is the identifier of the parent node of the node.
DeList is a list of the descendants of the port.

dia_out_port_node(DiagramId, PortId, InOutFlag, Level, Type1, Type2, ParentId, DeList)
DiagramId is the identifier of the diagram.
PortId is the identifier of the output port.
InOutFlag shows whether the node is input port node or output port node.
Level defines the level of this port within a composite port.
Type1 is AND, OR, or XOR.
Type2 is SINGLE, REPEAT, CONDITIONAL, R_C, or C_R.
ParentId is the identifier of the parent node of the node.
DeList is a list of the descendants of the port.
Any flow is a port node at Level 1, and its DeList is empty. The Level of a high level node is greater than 1, and the DeList is not empty since the node must have descendants.

dia_in_port_table(DiagramId, NodeList, PortIdNum)
DiagramId is the identifier of the diagram.
NodeList contains node identifiers. If the diagram has a complete port, then there is only one node within the NodeList.
PortIdNum is the unique name of the port within the port structure.

dia_out_port_table(DiagramId, NodeList, PortIdNum)
DiagramId is the identifier of the diagram.
NodeList contains node identifiers. If the diagram has a complete port, then there is only one node within the NodeList.
PortIdNum is the unique name of the port within the port structure.

process(ProcessId, Decom, FromList, ToList, Class, Des)
ProcessId is the identifier of the process.
Decom is the identifier of the (diagram of the) decomposition of the process.
FromList contains the identifiers of the flows coming into the process.
ToList contains the identifiers of the flows going out of the process.
Class may have the following values:
- 0: The process is still a PrM process.
- 1: The process has got a PLD description.
Des is a textual description.

in_port_node(ProcessId, PortId, Level, Type1, Type2, ParentId, DeList)
ProcessId is the identifier of the process.
PortId is the identifier of the input port.
Level defines the level of this port in a composite port.
Type 1 is AND, OR, or XOR.
Type 2 is SINGLE, REPEAT, CONDITIONAL, R_C, or C_R.
ParentId is the identifier of the parent node of the node.
DeList is a list of the descendants of the port.
Any flow is a port node at Level 1, and its DeList is empty. The Level of a high level node is greater than 1, and the DeList is not empty since the node must have descendants.

out_port_node(ProcessId, PortId, Level, Type1, Type2, ParentId, DeList)

ProcessId is the identifier of the process.
PortId is the identifier of the output port.
Level defines the level of this port in a composite port.
Type1 is AND, OR, or XOR.
Type2 is SINGLE, REPEAT, CONDITIONAL, R_C, or C_R.
ParentId is the identifier of the parent node of the node.
DeList is a list of the descendants of the port.
Any flow is a port node at Level 1, and its DeList is empty. The Level of a high level node is greater than 1, and the DeList is not empty since the node must have descendants.

in_port_table(ProcessId, NodeList, PortIdNum)

ProcessId is the identifier of the process.
NodeList contains node identifiers. If the process has a complete port, then there is only one node within NodeList.
PortIdNum is the name of the new port identifier.

out_port_table(ProcessId, NodeList, PortIdNum)

ProcessId is the identifier of the process.
NodeList contains node identifiers. If the process has a complete port, then there is only one node within NodeList.
PortIdNum is the name of the new port identifier.

store(StoreId, FromList, ToList, Class, Des)

StoreId is the identifier of the store.
FromList contains the identifiers of the flows coming into the store.
ToList contains the identifiers of the flows going out of the store.
Class is used to specify the type of the store.
Des is a textual description.

agent(AgentId, FromList, ToList, Class, Des)

AgentId is the identifier of the agent.
FromList contains the identifiers of the flows coming into the agent.
ToList contains the identifiers of the flows going out of the agent.
Class is used to specify the type of the agent.
Des is a textual description.
timer($\text{TimerId, FromList, ToList, Class, Des}$)
$\text{TimerId}$ is the identifier of the timer.
$\text{FromList}$ contains the identifiers of the flows coming into the timer.
$\text{ToList}$ contains the identifiers of the flows going out of the timer.
$\text{Class}$ is used to specify the type of the timer.
$\text{Des}$ is a textual description.

ex_in($\text{ExInId, FromList, Class, Des}$)
$\text{ExInId}$ is the identifier for the external input flow.
$\text{FromList}$ contains the identifiers of the flows coming from the external input.
$\text{Class}$ is the type of the flow.
$\text{Des}$ is a textual description.

ex_out($\text{ExOutId, ToList, Class, Des}$)
$\text{ExOutId}$ is the identifier for the external output flow.
$\text{ToList}$ contains the identifiers of the flows going into the external input.
$\text{Class}$ is the type of the flow.
$\text{Des}$ is a textual description.

flow($\text{FlowId, From, To, Property, Text}$)
$\text{FlowId}$ is the identifier of the flow.
$\text{From}$ indicate the source of the flow.
$\text{To}$ indicate the destination of the flow.
$\text{Property}$ may have the following values:
- 0: “non-terminate, non-trigger”
- 1: “non-terminate, trigger”
- 2: “terminate, non-trigger”
- 3: “terminate, trigger”
$\text{Text}$ is the name of the flow.

layout($\text{ObjectId, X, Y, Dots, Class}$)
$\text{ObjectId}$ is the identifier of the object (process, store, etc.).
$\text{X}$ is the x-coordinate of the object in the diagram.
$\text{Y}$ is the y-coordinate of the object in the diagram.
$\text{Dots}$ is a list of all the dots linking the flow lines to and from the figure.
$\text{Class}$ represents the graphic properties of a figure.

flow_fig($\text{FlowId, DotList, FirstDot, LastDot}$)
$\text{FlowId}$ is the identifier of the flow.
$\text{DotList}$ is a list of all identifiers of the dots that link the lines of a flow.
$\text{FirstDot}$ is the identifier of the dot which starts the flow.
$\text{LastDot}$ is the identifier of the dot which ends the flow.
`dot(DotId, X, Y)`

- `DotId` is the identifier of the dot.
- `X` is the x-coordinate of the dot.
- `Y` is the y-coordinate of the dot.
G. PPP facts of a detailed example

This appendix presents the PPP facts of the detailed example presented in section 7.4.3. The facts reflect the Data Flow diagrams depicted in figures 5-4, 5-11, and 5-12 respectively.

The set of PPP facts that is necessary to represent these three Data Flow diagrams is depicted in figure G-1, G-2, and G-3 respectively.

```plaintext
prm_diagram
(d1000, -, -, (p1001,p1002,p1003), (s1001,s1002,s1003), 
(a1001,a1002), -, -, -, (f1001,f1002,f1003,f1004,f1005,f1006,f1007,f1008))
process
(p1001, p1010, (f1001,f1002,f1003), (f1004), 0, "Estimate future need")
process
(p1002, p1020, (f1004,f1005), (f1006,f1007), 0, "Find best supplier")
process
(p1003, -, (f1006,f1007), (f1008), 0, "Place order")
store
(s1001, -, (f1003), 0, "Back orders")
store
(s1002, -, (f1002), 0, "Order history")
store
(s1003, -, (f1005), 0, "Supplier data")
agent
(a1001, -, (f1001), 0, "Stock clerk")
agent
(a1002, (f1008), -, 0, "Supplier")
flow
(f1001, a1001, p1001, 1, "Replenish part")
flow
(f1002, s1002, p1001, 0, "Order history")
flow
(f1003, s1001, p1001, 0, "Back orders")
flow
(f1004, p1001, p1002, 3, "Order quantity")
flow
(f1005, s1003, p1002, 0, "Supplier data")
flow
(f1006, p1002, p1003, 0, "Chosen supplier")
flow
(f1007, p1002, p1003, 3, "Order quantity")
flow
(f1008, p1002, a1002, 2, "Order")
layout
(a1001, 100, 550, (dt10001), 0)
layout
(a1002, 1400, 550, (dt10013), 0)
layout
(p1001, 500, 600, (dt10002,dt10003,dt10004,dt10005), 0)
layout
(p1002, 850, 600, (dt10006,dt10007,dt10008,dt10009), 0)
layout
(p1003, 1150, 600, (dt10010,dt10011,dt10012), 0)
layout
(s1001, 450, 200, (dt10016), 0)
layout
(s1002, 400, 100, (dt10014), 0)
layout
(s1003, 750, 250, (dt10018), 0)
flow_fig
(f1001, (dt10001,dt10002), (dt10001), (dt10002))
flow_fig
(f1002, (dt10014,dt10015,dt10003), (dt10014), (dt10003))
flow_fig
(f1003, (dt10016,dt10017,dt10004), (dt10016), (dt10004))
flow_fig
(f1004, (dt10005,dt10006), (dt10005), (dt10006))
flow_fig
(f1005, (dt10018,dt10019,dt10007), (dt10018), (dt10007))
flow_fig
(f1006, (dt10008,dt10010), (dt10008), (dt10010))
flow_fig
(f1007, (dt10009,dt10011), (dt10009), (dt10011))
flow_fig
(f1008, (dt10012,dt10013), (dt10012), (dt10013))
dot
(dt10001, 300, 500)
dot
(dt10002, 500, 500)
dot
(dt10003, 500, 450)
dot
(dt10004, 500, 400)
dot
(dt10005, 650, 450)
```
Fig. G-1: PPP facts for Replenish Stock 1.2
Appendix G. PPP facts of a detailed example

```
dot(dt10106, 550, 450)
dot(dt10107, 100, 150)
dot(dt10108, 250, 150)
dot(dt10109, 400, 200)
dot(dt10110, 500, 200)
dot(dt10111, 550, 430)
dot(dt10112, 400, 150)
dot(dt10113, 900, 150)
dot(dt10114, 700, 450)
dot(dt10115, 800, 450)
dot(dt10116, 900, 200)
dot(dt10117, 1050, 180)
dot(dt10118, 1250, 180)
dot(dt10119, 500, 430)
dot(dt10120, 800, 200)
```

Fig. G-2: PPP facts for Estimate Future Need 1.0

```
prm_diagram(d1020, d1000, p1002, (p1021,p1022,p1023), -, -, -(f1004,f1005),
            (f1006,f1007), -, ((f1021,f1022,f1023,f1024,f1025,f1026,f1027))

process(p1021, -, (f1022), (f1023), 0, "Check supplier history")
process(p1022, -, (f1021,f1023), (f1024,f1025), 0, "Check ability to deliver")
process(p1023, -, (f1024,f1025), (f1026,f1027), 0, "Compare price")
ex_in(f1004, f1021, 1, "Order quantity")
ex_in(f1005, f1022, 1, "Supplier data")
ex_out(f1006, f1026, 3, "Order quantity")
ex_out(f1007, f1027, 0, "Chosen supplier")
flow(f1021, f1004, p1022, 1, "Order quantity")
flow(f1022, f1005, p1021, 1, "Supplier data")
flow(f1023, p1021, p1022, 2, "Good track record")
flow(f1024, p1022, p1023, 3, "Order quantity")
flow(f1025, p1022, p1023, 0, "Able to deliver")
flow(f1026, p1023, f1006, 3, "Order quantity")
flow(f1027, p1023, f1007, 0, "Chosen supplier")
layout(p1021, 250, 250, (dt10204,dt10205), 0)
layout(p1022, 600, 470, (dt10202,dt10208,dt10209,dt10211), 0)
layout(p1023, 900, 470, (dt10210,dt10212,dt10213,dt10215), 0)
flow_fig(f1021, (dt10201,dt10202), dt10201, dt10202)
flow_fig(f1022, (dt10203,dt10204), dt10203, dt10204)
flow_fig(f1023, (dt10205,dt10206,dt10207,dt10208), dt10205, dt10208)
flow_fig(f1024, (dt10209,dt10210), dt10209, dt10210)
flow_fig(f1025, (dt10211,dt10212), dt10211, dt10212)
flow_fig(f1026, (dt10213,dt10214), dt10213, dt10214)
flow_fig(f1027, (dt10215,dt10216), dt10215, dt10216)
dot(dt10201, 100, 400)
dot(dt10202, 600, 400)
dot(dt10203, 100, 170)
dot(dt10204, 250, 170)
```
Fig. G-3: PPP facts for *Find Best Supplier 1.1*
H. Object structure of a detailed example

This appendix presents the detailed object structure of the detailed example presented in section 7.4.3. The object structure has been derived according to the steps given in the procedure to derive specific object structures that was presented in section 7.4.2. The input data for the object structure generation has been the PPP facts given in appendix G.

<table>
<thead>
<tr>
<th>Object_id</th>
<th>d1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>“Replenish Stock”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>N.A.</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A.</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>prm_diagram</td>
<td>(d1000, -,-, (p1001,p1002,p1003), (s1001,s1002,s1003), (a1001,a1002), -,-,-, (f1001,f1002,f1003,f1004,f1005,f1006,f1007,f1008))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>p1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>“Estimate future need”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.0</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>d1010;1.0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>(p1001, p1010, (f1001,f1002,f1003), (f1004), 0, “Estimate future need”)</td>
</tr>
<tr>
<td>layout</td>
<td>(p1001, 500, 600, (dt10002,dt10003,dt10004,dt10005), 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>p1002</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>“Find best supplier”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.1</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>d1020;1.1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>(p1002, p1020, (f1004,f1005), (f1006,f1007), 0, “Find best supplier”)</td>
</tr>
<tr>
<td>layout</td>
<td>(p1002, 850, 600, (dt10006,dt10007,dt10008,dt10009), 0)</td>
</tr>
<tr>
<td>Object_id</td>
<td>p1003</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Object_name</td>
<td>“Place Order”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.0</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A.</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>process(p1003, - (f1006,f1007), (f1008), 0, “Place order“)</td>
<td></td>
</tr>
<tr>
<td>layout(p1003, 1150, 600, (dt10010,dt10011,dt10012), 0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>s1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>“Back orders”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>store(s1001, - (f1003), 0, “Back orders“)</td>
<td></td>
</tr>
<tr>
<td>layout(s1001, 450, 200, (dt10016), 0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>s1002</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>“Order history”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>store(s1002, - (f1002), 0, “Order history“)</td>
<td></td>
</tr>
<tr>
<td>layout(s1002, 400, 100, (dt10014), 0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>s1003</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>“Supplier data”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>store(s1003, - (f1005), 0, “Supplier data“)</td>
<td></td>
</tr>
<tr>
<td>layout(s1003, 750, 250, (dt10018), 0)</td>
<td></td>
</tr>
<tr>
<td>Object_id</td>
<td>a1001</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>Object_name</td>
<td>&quot;Stock clerk&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
</tbody>
</table>

**agent**(a1001, -, (f1001), 0, "Stock clerk")

**layout**(a1001, 100, 550, (dt10001), 0)

<table>
<thead>
<tr>
<th>Object_id</th>
<th>a1002</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>&quot;Supplier&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
</tbody>
</table>

**agent**(a1002, (f1008), -, 0, "Supplier")

**layout**(a1002, 1400, 550, (dt10013), 0)

<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>&quot;Replenish part&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
</tbody>
</table>

**flow**(f1001, a1001, p1001, 1, "Replenish part")

**flow**(_fig_(f1001, (dt10001, dt10002), dt10001, dt10002)

**dot**(dt10001, 300, 500)

**dot**(dt10002, 500, 500)
<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1002</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>&quot;Order history&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
</tbody>
</table>

```latex
\text{flow}(f1002, s1002, p1001, 0, "Order history")
\text{flow}\_\text{fig}(f1002, (dt10014,dt100dt10015,dt10003), dt10014, dt10003)
dot(dt10014, 400, 100)
dot(dt10015, 400, 450)
dot(dt10003, 500, 450)
```

<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1003</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>&quot;Back orders&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
</tbody>
</table>

```latex
\text{flow}(f1003, s1001, p1001, 0, "Back orders")
\text{flow}\_\text{fig}(f1003, (dt10016,dt10017,dt10004), dt10016, dt10004)
dot(dt10016, 450, 200)
dot(dt10017, 450, 400)
dot(dt10004, 500, 400)
```

<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>&quot;Order quantity&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
</tbody>
</table>

```latex
\text{flow}(f1004, p1001, p1002, 3, "Order quantity")
\text{flow}\_\text{fig}(f1004, (dt10005,dt10006), dt10005, dt10006)
dot(dt10005, 650, 450)
dot(dt10006, 850, 450)
```
### Object structure of a detailed example

<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1005</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>&quot;Supplier data&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flow(f1005, s1003, p1002, 0, &quot;Supplier data&quot;)</td>
</tr>
<tr>
<td></td>
<td>flow_fig(f1005, (dt10018,dt10019,dt10007), dt10018, dt10007)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10018, 750, 250)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10019, 750, 400)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10007, 850, 400)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1006</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>&quot;Chosen supplier&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flow(f1006, p1002, p1003, 0, &quot;Chosen supplier&quot;)</td>
</tr>
<tr>
<td></td>
<td>flow_fig(f1006, (dt10008,dt10010), dt10008, dt10010)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10008, 1000, 450)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10010, 1150, 450)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1007</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>&quot;Order quantity&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flow(f1007, p1002, p1003, 3, &quot;Order quantity&quot;)</td>
</tr>
<tr>
<td></td>
<td>flow_fig(f1007, (dt10009,dt10011), dt10009, dt10011)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10009, 1000, 400)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10011, 1150, 400)</td>
</tr>
<tr>
<td>Object_id</td>
<td>f1008</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Object_name</td>
<td>&quot;Order&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.2</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1000;1.2</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>flow</td>
<td>(f1008, p1002, a1002, 2, &quot;Order&quot;)</td>
</tr>
<tr>
<td>flow_fig</td>
<td>(f1008, (dt10012,dt10013), dt10012, dt10013)</td>
</tr>
<tr>
<td>dot</td>
<td>(dt10012, 1300, 450)</td>
</tr>
<tr>
<td></td>
<td>(dt10013, 1400, 450)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>d1010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>&quot;Estimate future need&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.0</td>
</tr>
<tr>
<td>Part_of</td>
<td>N.A</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>prm_diagram</td>
<td>(d1010, d1000, p1001, (p1011,p1012,p1013), (f1001,f1002,f1003), (f1004), (f1011,f1012,f1013,f1014,f1015,f1016,f1017))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>p1011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>&quot;Check back orders&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.0</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1010;1.0</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>(p1011, (f1013), (f1014,f1015), 0, &quot;Check back orders&quot;)</td>
</tr>
<tr>
<td>layout</td>
<td>(p1011, 250, 280, (dt10108,dt10109,dt10112), 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>p1012</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>&quot;Avg. need last year&quot;</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.0</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1010;1.0</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>(p1012, (f1011,f1012,f1014), (f1016), 0, &quot;Avg. need last year&quot;)</td>
</tr>
<tr>
<td>layout</td>
<td>(p1012, 550, 550, (dt10102,dt10106,dt10111,dt10114), 0)</td>
</tr>
<tr>
<td><strong>Object_id</strong></td>
<td>p1013</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>Object_name</strong></td>
<td>&quot;Calc. present need&quot;</td>
</tr>
<tr>
<td><strong>Version_no</strong></td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Part_of</strong></td>
<td>d1010;1.0</td>
</tr>
<tr>
<td><strong>Solved_by</strong></td>
<td>N.A</td>
</tr>
<tr>
<td><strong>Owner</strong></td>
<td>User-1</td>
</tr>
<tr>
<td><strong>Transaction_id</strong></td>
<td>Trans-1</td>
</tr>
<tr>
<td><strong>Content</strong></td>
<td>process(p1013, -, (f1016,f1015), (f1017), 0, &quot;Calc. present need&quot;)</td>
</tr>
<tr>
<td></td>
<td>layout(p1013, 900, 280, (dt10116,dt10113,dt10117), 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Object_id</strong></th>
<th>f1011</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object_name</strong></td>
<td>&quot;Replenish part&quot;</td>
</tr>
<tr>
<td><strong>Version_no</strong></td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Part_of</strong></td>
<td>d1010;1.0</td>
</tr>
<tr>
<td><strong>Solved_by</strong></td>
<td>N.A</td>
</tr>
<tr>
<td><strong>Owner</strong></td>
<td>User-1</td>
</tr>
<tr>
<td><strong>Transaction_id</strong></td>
<td>Trans-1</td>
</tr>
<tr>
<td><strong>Content</strong></td>
<td>flow(f1011, f1001, p1012, 3, &quot;Replenish part&quot;)</td>
</tr>
<tr>
<td></td>
<td>ex_in(f1001, f1011, 3, &quot;Replenish part&quot;)</td>
</tr>
<tr>
<td></td>
<td>flowFig(f1011, (dt10101,dt10102), dt10101, dt10102)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10101, 100, 480)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10102, 550, 480)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Object_id</strong></th>
<th>f1012</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object_name</strong></td>
<td>&quot;Order history&quot;</td>
</tr>
<tr>
<td><strong>Version_no</strong></td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Part_of</strong></td>
<td>d1010;1.0</td>
</tr>
<tr>
<td><strong>Solved_by</strong></td>
<td>N.A</td>
</tr>
<tr>
<td><strong>Owner</strong></td>
<td>User-1</td>
</tr>
<tr>
<td><strong>Transaction_id</strong></td>
<td>Trans-1</td>
</tr>
<tr>
<td><strong>Content</strong></td>
<td>flow(f1012, f1002, p1012, 0, &quot;Order history&quot;)</td>
</tr>
<tr>
<td></td>
<td>ex_in(f1002, f1012, 0, &quot;Order history&quot;)</td>
</tr>
<tr>
<td></td>
<td>flowFig(f1012, (dt10103,dt10104,dt10105,dt10106), dt10103, dt10106)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10103, 100, 300)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10104, 330, 300)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10105, 330, 450)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10106, 550, 450)</td>
</tr>
</tbody>
</table>
Object_id   f1013
Object_name "Back orders"
Version_no  1.0
Part_of     d1010;1.0
Solved_by   N.A
...
Owner       User-1
Transaction_id Trans-1

Content
flow(f1013, f1003, p1011, 1, "Back orders")
ex_in(f1003, f1013, 1, "Back orders")
flow_fig(f1013, (dt10107,dt10108), dt10107, dt10108)
dot(dt10107, 100, 150)
dot(dt10108, 250, 150)

Object_id   f1014
Object_name "Orders pending"
Version_no  1.0
Part_of     d1010;1.0
Solved_by   N.A
...
Owner       User-1
Transaction_id Trans-1

Content
flow(f1014, p1011, p1012, 0, "Orders pending")
flow_fig(f1014, (dt10109,dt10110,dt10119,dt10111), dt10109, dt10111)
dot(dt10109, 400, 200)
dot(dt10110, 500, 200)
dot(dt10119, 500, 430)
dot(dt10111, 550, 430)

Object_id   f1015
Object_name "Orders pending"
Version_no  1.0
Part_of     d1010;1.0
Solved_by   N.A
...
Owner       User-1
Transaction_id Trans-1

Content
flow(f1015, p1011, p1013, 2, "Orders pending")
flow_fig(f1015, (dt10112,dt10113), dt10112, dt10113)
dot(dt10112, 400, 150)
dot(dt10113, 900, 150)
### Appendix H. Object structure of a detailed example

<table>
<thead>
<tr>
<th><strong>Object_id</strong></th>
<th>f1016</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object_name</strong></td>
<td>“Avg. need”</td>
</tr>
<tr>
<td><strong>Version_no</strong></td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Part_of</strong></td>
<td>d1010;1.0</td>
</tr>
<tr>
<td><strong>Solved_by</strong></td>
<td>N.A.</td>
</tr>
<tr>
<td><strong>Owner</strong></td>
<td>User-1</td>
</tr>
<tr>
<td><strong>Transaction_id</strong></td>
<td>Trans-1</td>
</tr>
</tbody>
</table>

**Content**

```
flow(f1016, p1012, p1013, 3, “Avg. need“)
flow_fig(f1016, (dt10114,dt10115,dt101120,dt101116), dt10114, dt10116)
dot(dt10114, 700, 450)
dot(dt10115, 800, 450)
dot(dt10120, 800, 200)
dot(dt10116, 900, 200)
```

<table>
<thead>
<tr>
<th><strong>Object_id</strong></th>
<th>f1017</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object_name</strong></td>
<td>“Order quantity”</td>
</tr>
<tr>
<td><strong>Version_no</strong></td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Part_of</strong></td>
<td>d1010;1.0</td>
</tr>
<tr>
<td><strong>Solved_by</strong></td>
<td>N.A.</td>
</tr>
<tr>
<td><strong>Owner</strong></td>
<td>User-1</td>
</tr>
<tr>
<td><strong>Transaction_id</strong></td>
<td>Trans-1</td>
</tr>
</tbody>
</table>

**Content**

```
flow(f1017, p1013, f1004, 3, “Order quantity“)
ex_out(f1004, f1017, 3, “Order quantity“)
flow_fig(f1017, (dt10117,dt10118), dt10117, dt10118)
dot(dt10117, 1050, 180)
dot(dt10118, 1250, 180)
```

<table>
<thead>
<tr>
<th><strong>Object_id</strong></th>
<th>d1020</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object_name</strong></td>
<td>“Find best supplier”</td>
</tr>
<tr>
<td><strong>Version_no</strong></td>
<td>1.1</td>
</tr>
<tr>
<td><strong>Part_of</strong></td>
<td>N.A.</td>
</tr>
<tr>
<td><strong>Solved_by</strong></td>
<td>N.A.</td>
</tr>
<tr>
<td><strong>Owner</strong></td>
<td>User-1</td>
</tr>
<tr>
<td><strong>Transaction_id</strong></td>
<td>Trans-1</td>
</tr>
</tbody>
</table>

**Content**

```
prm_diagram(d1020, d1000, p1002, (p1021,p1022,p1023), -, -, -,
(f1004,f1005), (f1006,f1007), -, (f1021,f1022,f1023,f1024,f1025,f1026,f1027))
```
<table>
<thead>
<tr>
<th>Object_id</th>
<th>p1021</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object_name</strong></td>
<td>“Check supplier history”</td>
</tr>
<tr>
<td><strong>Version_no</strong></td>
<td>1.1</td>
</tr>
<tr>
<td><strong>Part_of</strong></td>
<td>d1020;1.1</td>
</tr>
<tr>
<td><strong>Solved_by</strong></td>
<td>N.A</td>
</tr>
<tr>
<td><strong>Owner</strong></td>
<td>User-1</td>
</tr>
<tr>
<td><strong>Transaction_id</strong></td>
<td>Trans-1</td>
</tr>
<tr>
<td><strong>Content</strong></td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>p1021, - , (f1022), (f1023), 0, “Check supplier history“)</td>
</tr>
<tr>
<td>layout</td>
<td>p1021, 250, 250, (dt10204,dt10205), 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>p1022</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object_name</strong></td>
<td>“Check ability to deliver”</td>
</tr>
<tr>
<td><strong>Version_no</strong></td>
<td>1.1</td>
</tr>
<tr>
<td><strong>Part_of</strong></td>
<td>d1020;1.1</td>
</tr>
<tr>
<td><strong>Solved_by</strong></td>
<td>N.A</td>
</tr>
<tr>
<td><strong>Owner</strong></td>
<td>User-1</td>
</tr>
<tr>
<td><strong>Transaction_id</strong></td>
<td>Trans-1</td>
</tr>
<tr>
<td><strong>Content</strong></td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>p1022, - , (f1021,f1023), (f1024,f1025), 0, “Check ability to deliver“)</td>
</tr>
<tr>
<td>layout</td>
<td>p1022, 600, 470, (dt10202,dt10208,dt10209,dt10211), 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>p1023</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object_name</strong></td>
<td>“Compare price”</td>
</tr>
<tr>
<td><strong>Version_no</strong></td>
<td>1.1</td>
</tr>
<tr>
<td><strong>Part_of</strong></td>
<td>d1020;1.1</td>
</tr>
<tr>
<td><strong>Solved_by</strong></td>
<td>N.A</td>
</tr>
<tr>
<td><strong>Owner</strong></td>
<td>User-1</td>
</tr>
<tr>
<td><strong>Transaction_id</strong></td>
<td>Trans-1</td>
</tr>
<tr>
<td><strong>Content</strong></td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>p1023, - , (f1024,f1025), (f1026,f1027), 0, “Compare price“)</td>
</tr>
<tr>
<td>layout</td>
<td>p1023, 900, 470, (dt10210,dt10212,dt10213,dt10215), 0</td>
</tr>
</tbody>
</table>
Appendix H. Object structure of a detailed example

<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1021</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>“Order quantity”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.1</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1020;1.1</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td>flow(f1021, f1004, p1022, 1, “Order quantity“)</td>
</tr>
<tr>
<td></td>
<td>ex_in(f1004, f1021, 1, “Order quantity“)</td>
</tr>
<tr>
<td></td>
<td>flow_fig(f1021, (dt10201,dt10202), dt10201, dt10202)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10201, 100, 400)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10202, 600, 400)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1022</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>“Supplier data”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.1</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1020;1.1</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td>flow(f1022, f1005, p1021, 1, “Supplier data“)</td>
</tr>
<tr>
<td></td>
<td>ex_in(f1005, f1022, 1, “Supplier data“)</td>
</tr>
<tr>
<td></td>
<td>flow_fig(f1022, (dt10203,dt10204), dt10203, dt10204)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10203, 100, 170)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10204, 250, 170)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1023</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>“Good track record”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.1</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1020;1.1</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td>flow(f1023, p1021, p1022, 2, “Good track record“)</td>
</tr>
<tr>
<td></td>
<td>flow_fig(f1023, (dt10205,dt10206,dt10207,dt10208), dt10205, dt10208)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10205, 400, 170)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10206, 500, 170)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10207, 500, 370)</td>
</tr>
<tr>
<td></td>
<td>dot(dt10208, 600, 370)</td>
</tr>
<tr>
<td>Object_id</td>
<td>f1024</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>Object_name</td>
<td>“Order quantity”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.1</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1020;1.1</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>flow(f1024, p1022, p1023, 3, “Order quantity“)</td>
<td></td>
</tr>
<tr>
<td>flow_fig(f1024, (dt10209,dt10210), dt10209, dt10210)</td>
<td></td>
</tr>
<tr>
<td>dot(dt10209, 700, 400)</td>
<td></td>
</tr>
<tr>
<td>dot(dt10210, 900, 400)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1025</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>“Able to deliver”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.1</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1020;1.1</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>flow(f1025, p1022, p1023, 0, “Able to deliver“)</td>
<td></td>
</tr>
<tr>
<td>flow_fig(f1025, (dt10211,dt10212), dt10211, dt10212)</td>
<td></td>
</tr>
<tr>
<td>dot(dt10211, 700, 370)</td>
<td></td>
</tr>
<tr>
<td>dot(dt10212, 900, 370)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object_id</th>
<th>f1026</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_name</td>
<td>“Order quantity”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.1</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1020;1.1</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>flow(f1026, p1023, f1006, 3, “Order quantity“)</td>
<td></td>
</tr>
<tr>
<td>ex_out(f1006, f1026, 3, “Order quantity“)</td>
<td></td>
</tr>
<tr>
<td>flow_fig(f1026, (dt10213,dt10214), dt10213, dt10214)</td>
<td></td>
</tr>
<tr>
<td>dot(dt10213, 1000, 400)</td>
<td></td>
</tr>
<tr>
<td>dot(dt10214, 1250, 400)</td>
<td></td>
</tr>
<tr>
<td>Object_id</td>
<td>f1027</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Object_name</td>
<td>“Chosen supplier”</td>
</tr>
<tr>
<td>Version_no</td>
<td>1.1</td>
</tr>
<tr>
<td>Part_of</td>
<td>d1020;1.1</td>
</tr>
<tr>
<td>Solved_by</td>
<td>N.A</td>
</tr>
<tr>
<td>Owner</td>
<td>User-1</td>
</tr>
<tr>
<td>Transaction_id</td>
<td>Trans-1</td>
</tr>
</tbody>
</table>

**Content**

- flow(f1027, p1023, f1007, 0, “Chosen supplier“)
- ex_out(f1007, f1027, 0, “Chosen supplier“)
- flow_fig(f1027, (dt10215,dt10216), dt10215, dt10216)
- dot(dt10215, 1000, 370)
- dot(dt10216, 1250, 370)

Fig. 7-11: PPP objects for *Replenish Stock 1.2*

Observe that the objects that comprise the development object structure in figure 7-11 correspond to the objects in the component structure graph depicted in figure 5-13, with the added complexity arising from the decision to consider both the processes, the data stores, and the data flows to be the most fine grained objects of PrM.
I. The repository schema for the ERDBMS alternative

This appendix presents the database schema for a CWM front-end to PPP assuming the extended relational database solution has been chosen (section 7.4.4). The presentation follows the implementation of a prototype of the front-end utilising the TechRa extended RDBMS [91], as reported by Kimmel and Bjørnestad [88][20][21].

The prototype supports the following relations:

**Object(...)**

The object relation contains “core information” on the different objects, including the object contents.

**Object_link(...)**

The object link relation contains information on the relationships that exist among objects, e.g., objects that are version-related, objects that are merged from other objects, etc.

**Transaction(...)**

The transaction relation contains information on the development transactions.

**Transaction_object(...)**

The transaction_object relation contains information on which objects the transaction has checked out.

**Transaction_subscribe(...)**

The transaction_subscribe relation contains information on which objects the transaction subscribes.

**Contract(...)**

The contract relation contains information on check-out contracts.

**Person(...)**

The person relation contains information on the different users of the system.

We proceed to describe each relation in detail.

**Object(Object_name, Object_version_number, Creation_date, Author, Object_content)**

*Object_name* is the name of the object. For objects that are variant-related to another object the object name is concatenated with the variant name. For objects that are local versions, i.e. developed within a transaction, the identifier of the transaction is concatenated with the object name (and variant name).

*Object_version_number* is the version number of the version.

*Creation_date* is the time and date of creation of the version of the object.

*Author* is the identifier of the user that created the version of the object.
Object content is the object content defined as seq of byte.

Object link (Object name, Object version number, Link type, Link object name, Link object version number)

Object name is the name of the object.

Object version number is the version number of the version.

Link type indicates the type of the incoming relationship. The following values are legal:

- Revision of is used to indicate that the relationship in question is a relationship between an object (a conceptual view or a hierarchical object) identified by Object name and Object version number which is a revision of the object identified by Link object name and Link object version number.

- Variant of is used to indicate that the relationship in question is a relationship between an object (a conceptual view or a hierarchical object) identified by Object name and Object version number which is a variant of the object identified by Link object name and Link object version number.

- Checked out from is used to indicate that the relationship in question is a relationship between an object (a conceptual view or a hierarchical object) identified by Object name and Object version number which is checked out into a transaction from the object identified by Link object name and Link object version number.

- Checked back from is used to indicate that the relationship in question is a relationship between an object (a conceptual view or a hierarchical object) identified by Object name and Object version number which is checked in from a transaction from the object identified by Link object name and Link object version number.

- Local revision of is used to indicate that the relationship in question is a relationship between an object (a conceptual view or a hierarchical object) identified by Object name and Object version number which is a local revision of the object identified by Link object name and Link object version number, i.e. within a transaction.

- Local variant of is used to indicate that the relationship in question is a relationship between an object (a conceptual view or a hierarchical object) identified by Object name and Object version number which is a local variant of the object identified by Link object name and Link object version number, i.e. within a transaction.

- Built from is used to indicate that the relationship in question is a relationship between a conceptual view identified by Object name and Object version number which has been built from the conceptual view identified by Link object name and Link object version number.
- **Merged_from** is used to indicate that the relationship in question is a relationship between an object (a conceptual view or a hierarchical object) identified by *Object_name* and *Object_version_number* which is the result of a merging operation where the object identified by *Link_object_name* and *Link_object_version_number* is one of the operands.

- **Part_of** is used to indicate that the relationship in question is a relationship between a hierarchical object identified by *Object_name* and *Object_version_number* which is a hierarchical component of the object identified by *Link_object_name* and *Link_object_version_number*.

- **Solved_by** is used to indicate that the relationship in question is a relationship between two hierarchical objects where the object identified by *Object_name* and *Object_version_number* is realised by an object identified by *Link_object_name* and *Link_object_version_number* (note the analogy with procedure calls). The implicit dependency between a process and its decomposition (e.g., by using the same name) is in this way "maintained manually". Whenever a development object *B*, which is a decomposition of a process in another development object *A*, is modified to yield a new version of *B*, this does not automatically give a new version of *A*. Hence the use of the phrase "maintained manually".

The **Built_from**, **Merged_from**, **Variant_of**, and **Revision_of** are used to handle flat objects. The **Revision_of**, **Variant_of**, **Merged_from**, **Part_of**, and **Solved_by** are used to handle hierarchical objects. Note the symmetry between **Built_from** and **Variant_of**.

*Link_object_name* is the name of the object "at the other end of the link" specified by *Link_type*.

*Link_object_version_number* is the version number of the object "at the other end of the link" specified by *Link_type*.

**Transaction**(*Transaction_identifier*, *Publish*, *User_identifier*)

*Transaction_identifier* is the identifier of the development transaction.

*Publish* indicates whether the developer having checked out the object identified by *Contract_identifier* into this transaction wants to publish local versions of the object. i.e. whether other developers who want to subscribe to this object is allowed to do so.

*User_identifier* is the identifier of the developer which owns the transaction.

**Transaction_object**(*Transaction_identifier*, *Contract_identifier*)

*Transaction_identifier* is the identifier of the development transaction.

*Contract_identifier* is the identifier of the development contract belonging to the placeholder object that has been checked out by the transaction.

**Transaction_subscribe**(*Transaction_identifier*, *Subscribe_transaction*, *Subscribe_contract*)

*Transaction_identifier* is the identifier of the development transaction.
Subscribe_transaction is the identifier of the transaction that this transaction subscribes.

Subscribe_contract is the identifier of the contract belonging to the placeholder object that this transaction subscribes.

Contract(Contract_identifier, Object_name, Object_version_number, Transaction_identifier, Check-out_date, Status, Privilege)

Contract_identifier is the identifier of the development contract.

Object_name is the name of the placeholder for the object that has been checked out.

Object_version_number is the version number of the placeholder for the object that has been checked out.

Transaction_identifier is the identifier of the transaction that has checked out the placeholder object identified by Object_name and Object_version_number.

Check-out_date is the date and time of check-out, i.e. when the object was checked out into the transaction.

Status indicates the status of the object.

Privilege is the User_identifier of the developer responsible for the development object version. I.e. in the case of an update conflict who gets to say the last word.

Person(User_identifier, User_name, Email_address)

User_identifier is the identifier of the developer.

User_name is the name of the developer.

Email_address is the email address of the developer. The email is used to notify developers who have subscribed to an object that the object has been modified, i.e. that a new (local) version of the object has been created.
References


References


