# An Abstraction-Based Rule Approach to Large-Scale Information Systems Development

Anne Helga Seltveit
Email: ahs@idt.unit.no

Faculty of Electrical Engineering and Computer Science
The Norwegian Institute of Technology
N-7034 TRONDHEIM, NORWAY

**Abstract.** By introducing a rule language in IS development business rules can be explicitly represented. However, the use of rules implies that we have to find ways to deal with rule capture and complexity of large rule bases. The paper argues for an abstraction-based rule approach to information systems development. Simplifications of a specification are provided by allowing different views (abstractions) of the rule base and a framework for describing abstractions is suggested. A view is generated by a set of explicitly defined abstraction mechanisms, manually produced by a developer, or a combination of the two. Rather than operating on a full specification, relevant views can be applied at different stages of the development process (e.g., in rule capture) and then eventually integrated into a new version of the full specification. Views may be updated and multiple views are allowed to co-exist. Supporting multiple views and various versions of views demands for a versioning mechanism to keep track of the various versions of a full specification.

**Keywords: abstractions, abstraction-based development, structuring mechanisms, views, multiple views**

## 1 Introduction

Within IS Engineering, a large number of different modelling environments exists where most aim at supporting development of large data intensive, transaction-oriented information systems. Dealing with *massive amounts of information* is a key feature and the different approaches apply a variety of means (e.g., decomposition) to cope with the complexity of the application domain and thus, the specifications of the target IS. In spite of being able to express business rules explicitly by introducing a rule-based approach in IS development, we still have to cope with the same piece of reality. As pointed out in [14], *a specification paradigm cannot change an application's complexity, it can only move it from one level to another.*

Having this is mind, the use of rules implies that we have to find ways to deal with rule capture and complexity of large rule bases. We present an approach which explicitly addresses these problems, and the idea of abstractions is crucial in the approach taken. Unless facilities to abstract details of rules and facilities to

relate rules at different abstraction levels are developed, a rule-based approach to IS development may have little chance to succeed. Here the abstraction facilities together with a prescribed abstraction process are considered the key point in dealing with information (rules, objects, processes, etc.) at different abstraction levels in a systematical way.

To illustrate the practicality of the suggested approach, we look at an existing IS development environment, the TEMPORA environment. TEMPORA is a rule-based approach to IS development and aims at visibly maintaining the rules throughout the systems development process, from requirements specification through to an executable implementation. The paper argues how the TEMPORA environment can be extended with the abstraction facilities proposed here to deal with the complexity inherent in development of large industrial-sized applications.

The rest of the paper is structured as follows: Section 2 outlines the TEMPORA conceptual model. Section 3 proposes a way to characterize abstractions and Section 4 gives an overview of related work. Section 5 identifies requirements to an abstraction-based rule approach and describes structuring mechanisms. Section 6 introduces a set of abstraction mechanisms, and outlines how abstractions can be applied in building up IS specifications. Finally, Section 7 gives some concluding remarks and future directions.

## 2 The TEMPORA Conceptual Model

Since the abstraction approach presented in the next sections was developed to deal with the complexity of IS development using the TEMPORA conceptual languages, we will briefly describe the features of the languages and give an example which will form the basis for the reminder of the discussion of this paper.

The conceptual model has three components: the ERT Model [18, 31], the Process Model [3, 19], and the External Rule Language [17, 29, 31]. The ERT Model is an extended Entity Relationship Model and describes the static aspects of the real world, whereas the Process and Rule Models describe the dynamic aspects including the temporal dimension. Rules are also used to express constraints and derivations on the static model.

*Simplified Oil Processing Example* The oil processing unit of an offshore platform receives oil and gas together with water from the well and processes the well stream in a way which separates the different components from one another. Two of the major equipment components of the oil processing unit are separators and compressors. Separators are used to separate the oil phase from the gas phase whereas compressors are used to increase the pressure of the streams which are carried in various pipes. Based on information from the oil company, the operational conditions of the wellhead (e.g., pressure and temperature) are determined. These are together with information about compressor conditions, used as basis for choosing an appropriate number of separators and deciding

operational pressure of the separators. Separator conditions may also be revised as a result of simulations of the proposed design.

**The ERT Model** The static aspects of the real world is modelled by the ERT Model. The basic modelling constructs are: entity classes, relationships classes, and value classes. The model is also extended with constructs to describe complex objects, generalization/specialization, and temporal aspects. The graphical representation of the ERT Model constructs is given in Fig. 1 and an example illustrating the main features of the model is shown in Fig. 2. For a detailed description of the ERT Model, see [18, 31].

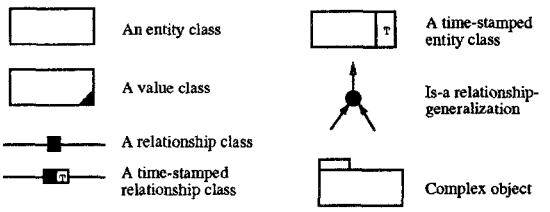


**Fig. 1.** Graphical representation of the ERT Model constructs.

**The Process Model** The Process Model, also called Process Interaction Diagrams (PID) is an extension of regular dataflow diagrams and is used to specify processes and their interaction in a formal way. This includes both the interactions between the processes at the same level of abstraction and how processes at any level of abstraction relate to their decompositions. The basic modelling concepts are: processes, stores, external agents, flows (may denote both control[1] and data flow), ports, and timers. An example of a Process Model is shown in Fig. 3. For a detailed description of the Process Model, see [3], [19] and [29].

**The External Rule Language** The External Rule Language (ERL) is a declarative rule language. It is based on first-order temporal logic and is extended with constructs for querying the ERT Model. Rules expressed using ERL may both decribe and constrain processes (of the PID) at any level of decomposition, but the model only requires them for describing the lowest level. In addition, the ERL is used to express constraints and derivations on the ERT Model.

All ERL rules are given a single general structure:

WHEN <trigger condition> IF <condition> THEN <conclusion>

where the WHEN and IF parts are optional.

[1] Also called *triggering* flow.

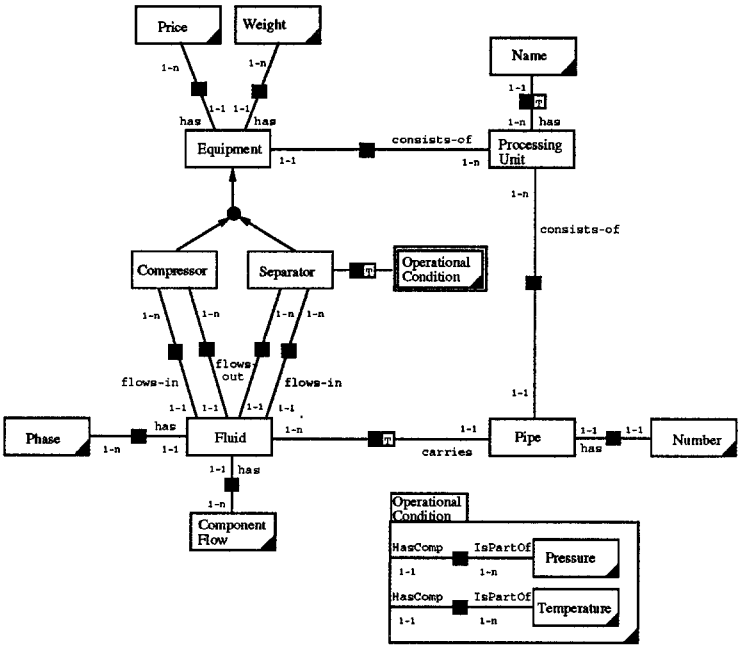


**Fig. 2.** ERT Model for oil processing design.

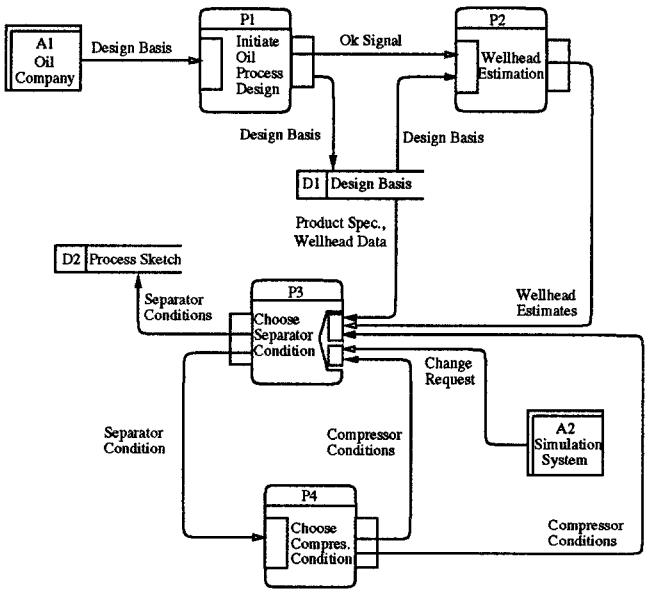


**Fig. 3.** PID Model for oil processing design.

The basic elements of ERL expressions are: a selection of data from the ERT, sets of data obtained from the ERT, and predicates which name tuples of data selected from the ERT. Compound expressions are constructed from basic elements by using connectives of classical first-order logic (e.g., **AND**) and temporal connectives (e.g., **SOMETIME_IN_PAST**).

To give procedural semantics to an ERL rule, a rule must be categorized as being a constraint rule, a derivation rule, or an action rule. A constraint rule expresses conditions of the ERT database which must not be violated. For example, ensuring that the number of separator steps is different from 2, can be expressed as:

**FOR_ALL** processing-unit (X) **IT_FOLLOWS_THAT COUNT** {Y **FOR_WHICH** processing-unit (X) consists-of separator (Y)} $\neq 2$

A derivation rule expresses how information can be derived from information that already exists. For example, deriving that equipment is associated with a high pressure process if the equipment is heavy and expensive, can be expressed as:

**IF** processing-unit (X) consists-of equipment (Y) [has weight = 'heavy', has price = 'expensive']
**THEN** high-pressure-process (Y)

An action rule expresses what actions to be taken if an event occurs and the conditions evaluate to true. For example, stating that upon receiving a change request provided that information about compressor conditions is available, new separator conditions will be computed, can be expressed as:

**WHEN** change-request (Separator, Temperature)
**IF** compressor-condition (Condition) **AND** calculate-pressure (Temperature, Condition, Pressure)
**THEN** separator-condition (Pressure) **AND** processing-unit consists-of separator (Separator) [has operational-conditions has pressure = Pressure]

For a detailed description of ERL, see [17, 29, 31].

**The coupling between the models** The relationship between the Rule Model and the Process Model is depicted in Fig. 4. Each non-decomposed process should have associated a set of ERL rules describing the behaviour of the process. In addition, one may optionally specify the behaviour of decomposed processes, these rules being interpreted as constraints on the behaviour of the rules describing the non-decomposed processes.

The relationship between the Rule Model and the Process Model as depicted in Fig. 4, can be described as follows. The trigger part is extracted from the process structure (i.e., triggering flow) and corresponds to the WHEN part of an ERL rule. The conditional part is extracted from the process structure (i.e.,

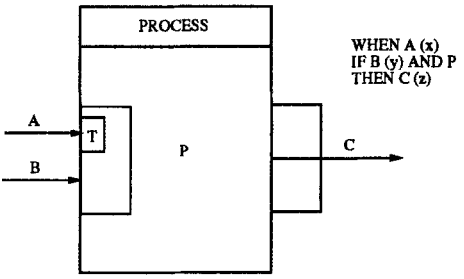


**Fig. 4.** Relationship between the Process and Rule Models.

non-triggering flow) and from the process logic (expressed in a subset of ERL). This corresponds to the IF part of an ERL rule. The action part, that is, the THEN part of an ERL rule is extracted from the process structure (i.e. output flows).

The Process Model provides an overall structure to the ERL rules [29]. The ERL rules are grouped in clusters according to the process they are associated with. A formal definition of the coupling of the Process Model and Rule Model is given in [13, 31].

The Process Model is given semantics by an underlying temporal model of the dynamic aspects of the specified system, and it is via these semantics that the connection between processes and rules can be made. An informal description of the semantics of the underlying model is provided in [13], and the interested reader is referred to [20] for a mathematically rigorous definition.

# 3 Characterizing Abstractions

## 3.1 The Concept of Abstraction

Abstraction can be defined as the process of *separating* relevant and irrelevant details from a context. The irrelevant details are suppressed whereas the relevant details are high-lighted. Accordingly, we aim at providing facilities for ensuring an adequate representation and presentation of the information at the appropriate level of abstraction at any time of development.

We distinguish between representation and presentation. Representation deals with the modelling formalisms used to describe the system. Presentation deals with how the system specifications are used as a means of communication among the actors involved. The former emphasizes expressiveness and formality whereas the latter emphasizes expressiveness and user-friendliness.

A presentation facility may use the entire representation formalism (e.g., ERL) or only subsets of it (e.g., simplifications of ERL) and most often a set of additional features are included to improve the readability and understanding of specifications (e.g., language conversion).

In this way, the abstraction problem may be considered two-fold; What details to be included (contents) and how these details are to be structured (layout).

When certain details are left out (i.e., abstracted away) this will change how the remaining details are structured.

### 3.2 Why We Introduce Abstractions

Basically there are two reasons for introducing abstractions:

1. To cope with the complexity of the Universe of Discourse (*domain dependence*).
2. To cope with the complexity of a particular specification language (*language dependence*).

In information systems development we have to deal with both cases. Developing information systems requires that we at some point in the development process must specify the details of the system. The complexity of the problem will decide where and when the details are to be considered. As more details are gathered and represented by appropriate modelling formalisms, the specifications may grow too large to be shown at one time or include details which may be of no interest to all the actors involved. Furthermore, different actors may hold slightly different views of the reality and the system to be built, and the final information system must accomodate a compromise of all these views. From the final model which is the most complex one, simpler views can be derived. Thus in general, we face the problem of deciding what is important at a particular time, that is, *what to show when to whom, in what form, and at what level of detail.*

Irrelevant of the modelling formalism used, the specifications become large. Having this in mind, we know that it is not sufficient to have a model which is meeting the requirements for expressiveness and formality unless it also provides some mechanisms to deal with the complexity as the size of the specifications grows. This together with the fact that formal languages with high expressive power become hard to understand are the rationale behind our search for systematical ways of performing simplifications to the TEMPORA conceptual languages.

### 3.3 Classifying Abstractions

We may classify abstractions at least in two dimensions according to (see Fig. 5):

- Specification level
- Number of details

A specific *abstraction level* A is determined by the tuple (X,Y), where X describes the number of details to be included at a particular specification level and Y describes what specification level(s) to focus at. For instance, an abstraction level may be located at the Business Modelling level showing all the top-level processes in the domain where all but triggering flows (control flows) are left out. This may be specified as follows:

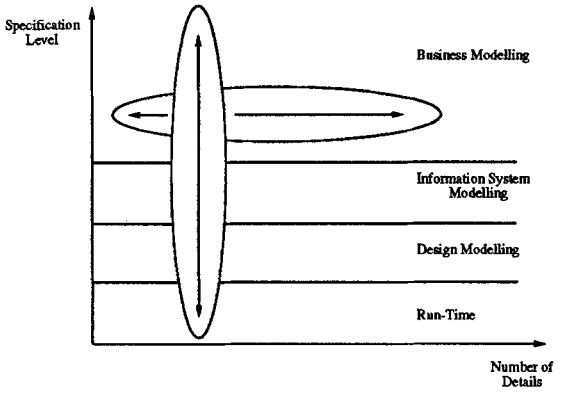


**Fig. 5.** Framework for describing abstractions.

*A (Business Modelling Level.PID.Level0, Control Flow)*

Hence we distinguish between two major types of abstractions: *horizontal abstractions* and *vertical abstractions*. Performing a horizontal abstraction on a specification means to decide what details to be included at a particular specification level and suppress all other information at that level (i.e., a horizontal abstraction computes X). A horizontal abstraction may involve:

- Removal of details (e.g., process hierarchies).
- Conversion towards other languages (e.g., paraphrazing).
- Combinations of the two.

By performing a vertical abstraction, we eliminate irrelevant details across specification levels (i.e., a vertical abstraction computes Y). It may consist of a set of vertical transformations of a system specification. Keeping the connections between corresponding/related components at different levels enables tracing of specifications across development stages.

Vertical and horizontal abstractions may be further divided into *application dependent abstractions* (e.g., applying relations between rules at different specification levels) and *model dependent abstractions* (e.g., removing complexity increasing constructs of a language).

## 4 Related Work

In Information Systems Engineering, the concept of abstraction or information hiding is a crucial one. In the widest sense, a model is an abstraction of the reality and a number of different specification languages are developed to cope with this reality. As we want to model even larger and more complex systems, specifications expressed using these languages become difficult to use and understand. To deal with the complexity, a variety of abstraction mechanisms

are proposed. From the database community, data modelling abstractions (such as generalization/specialization, aggregation, association, and classification [32]) and mechanisms to allow definition of different views (e.g., SQL views [9]) are well known means to abstract away irrelevant details. These abstraction mechanisms are also adopted by several specification languages for static modelling. For instance, the Phenomenon Model developed by Sølvberg supports generalization/specialization, aggregation, and association [27, 28] and introduces the concept of scenario for defining different views of a domain [28]. An overview of hierarchical constructs in static modelling languages is given in [26].

In modelling the dynamics of a system, decomposition is a widely known abstraction mechanism. In spite of being the subject of much debate (e.g., [8]) it has proven to be useful in practice and a number of methods are based on functional decomposition (e.g., DFD-like languages [10, 31]). An overview of hierarchical constructs in dynamic modelling languages is given in [26].

Functional decomposition has also been adopted by several object-oriented approaches to perform top-down specification of processes (e.g., [33]). Aggregation is another abstraction mechanism supported by object-oriented approaches. In addition, Wirfs-Brock et al. [34] has suggested the use of contracts and subsystems to group operations and objects at different levels of abstraction.

A striking feature however is that the idea of abstraction as a specification paradigm is not widespread. To our knowledge, the only abstraction-based software development approach is developed by V. Berzins et al. [5, 6, 16]. The key idea being to provide mechanisms called black box descriptions at different levels of abstraction which enable description of external behaviour of any system (and sub-systems) to be distinguished from the internal mechanisms that eventually are used to realize that behaviour. Thus, a complex system is described by a set of independent abstractions (black boxes) that are described, understood, and analyzed independently of the details that are used to implement the system. The concepts applied in the initial approach [5] are similar to abstraction types well known from programming languages and specification languages for information systems [4]: abstract data types, iterators, state-machines, and transformers. More recent work addresses the problems of information overload in rapid prototyping of large-scale real-time systems. A model based on Data Flow Diagrams augmented with multiple views is proposed where a distinction is made between facilities for providing overview pictures (summary views) and facilities for focusing on certain details (navigation structures and focused slices) [16].

In addition, some CASE tools today have built in certain abstractions in their tools but these have very little formal basis (mainly functions implemented in the tools). One example is the grouping function provided by the IEF CASE tool where entities and relationships can be grouped to form higher level objects. However, such an object has no own behaviour and can only be considered as a vague composition facility. Another example is the concept of scenario in RDD (Requirements Driven Development [1]) which supports the generation of different views of a textual specification and allows multiple views to co-exist.

The limitation of this approach is the lack of formal specifications.

# 5 Requirements to an Abstraction-Based Development Approach

## 5.1 Identifying Requirements

Our aim is first to provide a rule-based development environment which makes it possible to capture information when it appears naturally in the development process. Secondly, this information should be expressed at a desirable abstraction level instead of forcing e.g., a high-level business rule to be expressed in a concrete implementation-oriented representation too early in the development process. We may then ask what is a right abstraction level and what is required from the development environment (i.e., languages, methods, and tools) to support the process of arriving at the right abstraction level? Using the definitions from Section 3, to find the right abstraction level of a specification level means to determine the specification level(s) to focus at and the appropriate number of details to be included at that level (those levels). Thus providing a right abstraction level requires at least:

- *Languages*: Formalisms to express information at different abstraction levels (includes both X and Y dimensions). This is provided by the TEMPORA conceptual languages.
- *Structuring mechanisms*: Mechanisms to relate information within a specification level (includes X dimension) and across levels (includes Y dimension), that is, relate information at different levels of abstraction.
- *Abstraction mechanisms*: Mechanisms to perform scoping of composite specifications, that is, facilities to suppress certain details from a whole according to some explicitly defined criteria.

To support a rule-based IS approach based on abstractions we may add the following requirements:

- *Multiple views.* A view is an abstraction of a specification and multiple views of the same specification should be allowed.
- *Multiple views to co-exist.* Multiple views of the same specification should be allowed to exist simultanously in the system.
- *Updates of views.* It should be allowed to modify a view directly.
- *Synthesis of views.* The process of building up a new specification based on the former full specification and a set of updated views should be supported (e.g., syntactical merge and more advanced support).
- *Versioning mechanism.* The versioning system should provide the facilities to manage different views and versions of views.
- *Abstraction process description.* Guidelines/methods for how to apply abstractions in the modelling process (e.g., rule capture, structuring, retrieval, and merging of specifications) should be described.
- *Tools.* Comprehensive tool support should be provided.

### 5.2 The Importance of Structuring Mechanisms

To be able to use the conceptual models of TEMPORA on large and complex real world problems, we need to provide adequate structuring mechanisms. Experience from a comprehensive case study (the Sweden Post case study [30]) revealed that there is a need for a tighter coupling between the TEMPORA models and also mechanisms to impose structure on the flat rule base [22, 30].

The purpose of structuring mechanisms varies at different stages of development. In this work, we classify structuring mechanisms into two groups according to which level they are used at:

- structuring mechanisms used at *the specification level*
- structuring mechanisms used at *the execution level*

The former is basically oriented towards improving the *communication aspect* of specifications, whereas the latter is oriented towards increasing the *performance* of executable systems. Since the focus of this work is on the early stages of development and abstractions, only the problem of structuring at the specification level is dealt with.

The specification level comprises Business Modelling, IS Modelling, and Design Modelling. The main objective of the structuring mechanisms at the this level is to improve the modelling languages' ability to deal with large and complex real world problems (i.e., handle massive amounts of information). In particular, it should contribute to *bridge the gap between the various specification languages* and also *facilitate scoping of models.* Some models will only apply to a particular part of an organization, whilst others must be applicable all over the organization and at the same time be related to the models of the part. Thus, we require mechanisms to scope models and relate parts of the global and local models.

Structuring and abstraction are intertwined concepts. In TEMPORA, the structuring mechanisms constitute the "glue" between different parts of a specification whereas the abstraction mechanisms will provide facilities to perform the actual scoping of the specifications. Rule structuring is considered the process of *representing explicitly* the structure of a set of rules (how they relate to one another) whereas rule abstraction is considered the process of *presenting* relevant aspects of a specification by hiding irrelevant details. Thus, adequate structuring mechanisms are a prerequisite for supporting abstractions; to remove details we need to know how different details relate to one another. We envisage a structuring approach where the integration of the TEMPORA conceptual model is based on links with clearly defined semantics rather than pure name-links (i.e., names must correspond in the specifications). This is in contrast to contemporary CASE tools which mainly support browsing and navigation in specifications due to the limited name-links they support (syntactical basis). Consequently, such tools cannot exploit the potential of formal specifications (e.g., tool kits are limited to support drawing of diagrams).

In general, the structuring mechanisms provide the links which allow for grouping/structuring of information which is logically related. Structuring mech-

anisms at the specification level may be classified into three types, as depicted in Fig. 6:

- *Intra-model links.* This includes means for structuring information within a model (e.g., decomposition in PIDs).
- *Inter-model links.* This includes means for structuring information between models at the same level (e.g., the coupling between the Process Model and the Rule Model [13]).
- *Inter-level links.* This includes means for structuring information between models at different levels (e.g., relating rules at different specification levels).

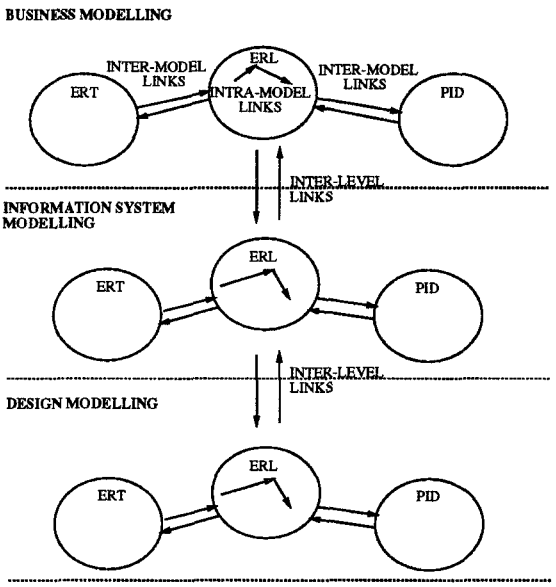


**Fig. 6.** Types of Structuring Mechanisms at the Specification Level.

We believe that the links described in the framework above provide all the links between different parts of a specification necessary to support scoping of TEMPORA specifications (abstractions). Details of the structuring mechanisms developed for the TEMPORA languages are described in [31].

## 6 Defining and Applying Multiple Views of an IS Specification

### 6.1 Defining Views of a Specification

A *view* is a selected part of a specification and is defined by a set of abstraction mechanisms (abstraction operators). The basic abstraction mechanisms can be

divided into three types which will be used as basis for automated support of scoping of specifications:

- Rule abstractions
- ERT abstractions
- PID abstractions

There are a number of ways to simplify a specification, each focusing on a different aspect of the system. Below an outline of abstractions in TEMPORA is given. The classification of abstractions into different abstraction types is of course highly *subjective.* The proposal is based on experience gained through case studies applying the TEMPORA conceptual languages (e.g., [30]) and the PPP languages and experience and requirements put forward by other researchers and practioners in the field.

Some critical voices may say that it would be sufficient to give the possibility for having a user-defined abstraction, without giving any directions to which specification details that should be abstracted away. However, experience so far has shown that there is a need to go about abstractions in a systematical manner to provide the user adequate tool support to avoid chaos when several simplications to the same specification are allowed.

**Defining Views Through Rule Abstractions** A rule base may be viewed from different perspectives through a set of relations. The relations relate rules in the rule base and each relation type corresponds to a specific perspective. Rules may be related to other rules in a number of ways: goal related, causal related, domain related, context related, exception related, etc. What relations to be included in a specification may differ in domains and applications. We envisage an approach where it should be easy to define new relation to be modelled. The relations are called inter-rule links which is a particular type of intra-model links (see previous section). The basic structure of an inter-rule link (i.e., a relation):

*<rule> <relation> <rule>*

The *rule* fields may be any ERL expression, and may represent an organization's goal, policy, action, plan, etc. The *relation* field may be any defined relation.

Experience using the TEMPORA approach indicates that there is a need to provide mechanisms to record at least goal related rules, causal related rules, and exception related rules in the development of data intensive, transaction-oriented information systems [31]. We have defined the following set of relations to relate rules in a TEMPORA rule base [23, 25, 26, 31]:

- **motivates** and **necessitates**
- **refers-to** and **causes**
- **overrules** and **suspends**

To illustrate the features of a rule abstraction an example showing the **motivates** and **necessitates** relations is given below. Details about each rule abstraction above are beyond the scope of this paper and we refer the interested reader to [24].

**Example: motivates and necessitates relations** The **motivates** and **necessitates** relations are introduced to describe hierarchies of purpose (correspond to goal hierarchies in organization theory). The relations express the purpose of one rule in terms of one or more rules at a higher level, that is, explain why the organization has a particular rule in terms of rules at the higher level. Given the rules A and B. A **motivates** B means that A is at a higher level than B and A gives the rationale behind B. By using the relation **necessitates**, an even stronger relationship between A and B can be expressed. A **necessitates** B indicates that A is at a higher level than B and the contents of B is necessary to achieve the contents of A. Thus, if A **necessitates** B we may also say that A **motivates** B (even though the opposite may not be true).

Fig. 7 shows an example hierarchy of purpose that includes both the **motivates** and **necessitates** relations.

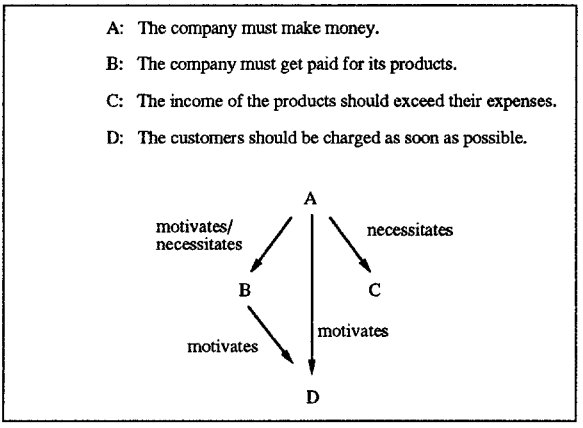


**Fig. 7.** An example hierarcy of purpose: the motivates and necessitates relations

**Defining Views Through ERT Abstractions** In a company we may divide information into global and local information. The former conveys information which there is some kind of consensus on throughout the organization, whereas the latter conveys local information used by a department, a group, or a person. As mentioned above, our development environment should be able to handle both global and local views (scoping) in a systematical manner. For instance, it should be easy to relate a local view to a global view. This is opening up for development of an enterprise-wide conceptual model and at the same time allowing for different perceptions of the business to co-exist. In our approach, this is achieved by defining a set of explicit ERT abstraction mechanisms and through the use of the scenario concept [28], here only the former is briefly described.

Main abstraction types associated with the ERT model, which are all contents related abstractions:

- Traditional data modelling abstractions. In TEMPORA, aggregation and generalization are supported.
- Construct abstractions. Constructs are removed from the specification such as value classes, is-a relationships, and time stamping of entity classes and relationship classes.
- Component abstractions. Point at certain entitiés and all their connections will be kept, and all other components are removed from the diagram.
- Grouping abstractions. Entities and relationships can be grouped to form higher level objects.

**Example: ERT abstraction** Fig. 8 shows an abstraction of Fig. 2 where the all value classes and timestamps of relationhip classes and entity classes are removed. An alternative notation is to show the value classes in a table associated with respective entity classes.

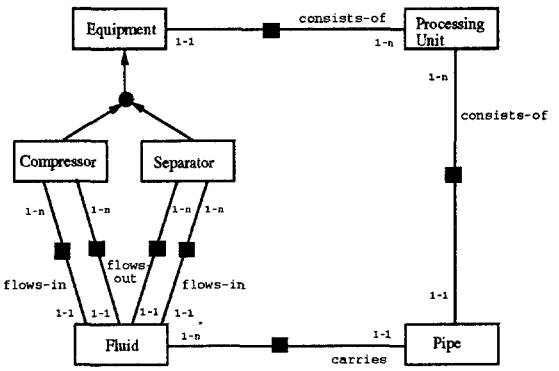


**Fig. 8.** ERT Model for oil processing design where value classes and timestamps are removed.

**Defining Views Through PID Abstractions** PID contains more details than conventional flow diagrams including all information flows and material flows in a system as well as their interrelationships in terms of input ports and output ports. Specifying systems beyond toy-examples implies that the number of details become large and the diagrams may become difficult to read and to understand. To mend this we suggest a number of abstractions, where each abstraction is a simplification of the model according to a specific criterion (a set of criteria):

Main abstraction types associated with the PID, which are all contents related abstractions:

- Port abstractions.
- Flow abstractions.
- Component abstractions.
- Control flow abstractions.
- Exception abstractions.
- Grouping abstractions.
- Process hierarchy abstractions.

We have chosen to divide abstractions into basic and auxiliary abstractions. Port and flow abstractions make up the basic abstractions whereas the rest is considered auxiliary abstractions. The basic abstractions are considered fundamental to handle the complexity of PID and should be implemented first. The auxiliary abstractions constitute a set of simplifications which seem to be advantageous but are not considered crucial for using PID.

To illustrate the features of a PID abstraction, a simple port abstraction is described below. Details about each abstraction above are beyond the scope of this paper and the interested reader is again referred to [24].

### Example: Port Abstractions

*Subclassifications*: strict port abstractions and relaxed port abstractions.

Performing a port abstraction removes the details of the ports from the specification and processes, flows, data stores, external agents, and timers remain. The notation for an abstracted port is a filled AND symbol. Fig. 9 shows the PID specification of the oil process (see Fig. 3) where the ports are abstracted away.

The ports may impose certain requirements on the location of flows entering or leaving a process, that is, where the conncection point between a flow and a process is located. Crossing flows may result. A more *relaxed* notation removes the port symbols from the specification and arrows may enter and leave a process at any edge (see Fig. 10). After abstracted the ports away from a specification, a messy diagram should be restructured (redrawn) to avoid crossing flows.

**Port abstraction algorithm (strict version):**

1. For each process of the diagram, substitute any number of simple and/or composite input ports (output ports) by *one* abstracted input port (output port).
2. Remove any duplicates of flows resulting from the abstraction of port details. This corresponds to a parallel flow abstraction.
3. Restructure diagram to avoid (reduce) crossing flows.

If a relaxed port abstraction is performed (i.e., ports are removed and arrows may enter and leave a port at any edge) and duplicates of flows are removed, the resulting diagram may resemble a traditional DFD.

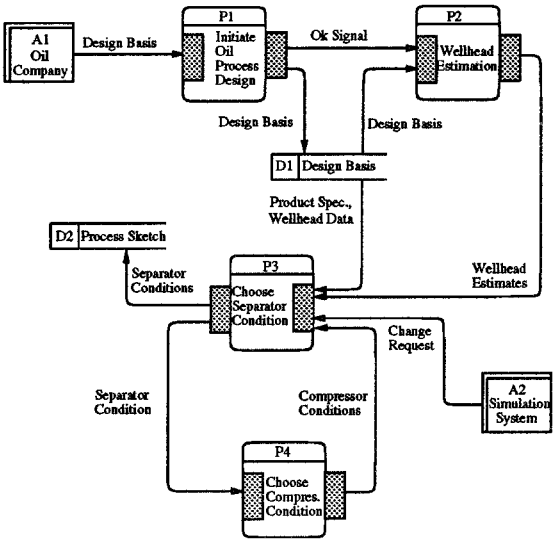


**Fig. 9.** PID specification of the oil process where the ports are abstracted away.

## 6.2 A Scenario of Abstraction-Based IS Development

**Different Types of Abstractions** The abstraction processes can be classified into three main types according to how abstractions are performed:

**Type 1** Simple abstractions.
**Type 2** Composed abstractions.
**Type 3** Update abstractions. This includes updates of type 1 and type 2.

**Type 1: Simple abstractions** Performing a simple abstraction means to perform any of the pre-defined abstraction mechanisms described in the previous section. Thus, the resulting view can be derived from the full specification at any time using the respective abstraction operator. A port abstraction is an example of a simple abstraction.

**Type 2: Composed abstractions** Performing a composed abstraction means that two or more simple abstractions are performed in consecutive order. An example of a composed abstraction is a a port abstraction followed by a flow abstraction on the same view.

**Type 3: Update abstractions** An update abstraction means a manually produced abstraction and is thus a modification of a specification/view which is not well defined (no abstraction operator defined). An example is a layout abstraction. Abstracting away certain details from a specification (e.g., ports and flows in a PID) simplifies the diagram and often, restructuring of the diagram

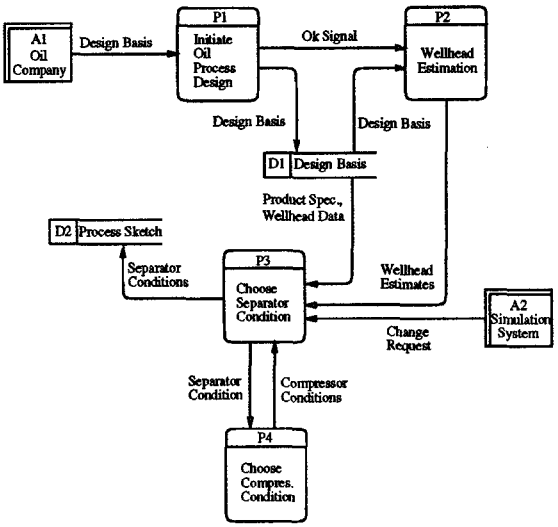


**Fig. 10.** PID specification of the oil process where the ports are removed.

is desirable (e.g., locations of symbols make the diagram too spacy or flows are unnecessarily crossing).

The restructuring of a diagram should not be considered a specific type of abstraction mechanism since the layout is the only difference between a restructured abstraction and the original one. A restructured view is treated as a *revision* of the original one and thus, it should be dealt with by the versioning system as a regular update of a view. Accordingly, we are allowed to keep different versions of the layout of a view and it is up to the analyst to decide how many layout versions of a view she actually wants to keep.

**Views Generated by Different Types of Abstractions** A view may be composed of information expressed using one or more modelling formalisms. By exploiting the coupling between the various languages (provided by the structuring mechanisms in Section 5), the lack of graphical notation and structuring features of the rule language can be compensated. In TEMPORA, this is essential in managing a large number of rules in a rule base. For instance, we may use the graphical representation of the ERT and PID languages to selects certain parts of a specification and derive the relevant associated rules. In this way, we may also say that we use the ERT and PID languages to define views of the rule base.

We may divide views into two major types:

- *Horizontal views.* A horizontal view is produced by a horizontal abstraction and consists of selected parts of a specification and optionally related specifications (e.g., expressed in another modelling language) from one specification level.

- *Vertical views.* A vertical view is produced by a vertical abstraction and consists of information from one or more specification levels.

Furthermore, horizontal and vertical views may be divided into *simple views* (produced by a simple abstraction), *composed views* (produced by a composed abstraction), and *updated views* (produced by an update abstraction).

### The Process of Building Up New Specifications

**The Merging Process** The process of putting a new revision of the full specification together will be referred to as the *merging* process. The following specifications may form the basis for this process, as depicted in Fig. 11:

- previous versions of the full specification (e.g., S1.1)
- updated versions of views of previous versions of the full specification (e.g., V1, V2, V3, and V4)

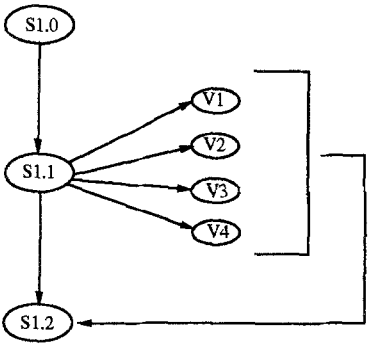


**Fig. 11.** Building a new revision based on updated views.

From a full specification S a number of views may be derived V1,..V*n* by applying any of the abstraction mechanisms described above. Derived and composed views of previous versions of a full specification are subsets of it and do not need to be considered explicitly in the merging process (i.e., already included in the full specification). Thus, updated views are the only views to be explicitly dealt with.

Allowing update abstractions means that we are able to modify views directly, and it is up to the developer to manipulate the view, no control is to be provided by the system. However, we have to provide facilities to support this activity. In particular, we have to address the problem of *inconsistencies between different views and the full specification from which the views originate.* Basically, we have to decide on the following:

- Should we allow for inconsistencies or
- Should we create a skeleton (also called a *placeholder*) for a new revision of the full specification as soon as a view is updated?

If we go for the former approach we only need to provide versioning of update abstractions and leave the process of integrating the new versions into the full specification to the developer. This does not seem to be the way to go due to the fact that providing this flexibility may cause more damage (chaos!) than support in the development process. Thus we have to come up with facilities to support the process of going from one full specification to another one by taking into account updated versions of views. A number of issues ought to be addressed:

- How can we create the placeholders?
- How should the process of building a new full specification take place?
- What kinds of support do we envision when building up a new full specification?
- What to be performed automatically/manually/supported in some way?
- What about the interplay with the version control system?
- How can we merge subschemas to create the full schema?
- How does this relate to integration of specifications developed by different people (i.e., the use of CSCW techniques in specification integration)?

A detailed description of each item above is beyond the scope of this paper and we will limit ourselves to present some support of the merging process below.

**Support of the Merging Process** The ideal situation would have been to allow updates on views and provide enough support to an automatical merge of relevant specifications. However, we cannot expect this process to be fully automatic but rather a mix of manual intervention and comprehensive computer support:

- Manually: manual inspection of specifications (full specifications and views) and relevant information is manually extracted and integrated in a new full specification (e.g., working styles may be virtual paper, Macintosh' Clipboard technique between specifications, ordinary 'cut and paste' between specifications, and "active" structures).
- Computer support: syntactical merge of different specifications, different ways of merging based on semantics of a specification, record modelling history (e.g., record what has happened to a view since checkout), group support (CSCW techniques to assist groups in the process of building up a new full specification e.g., synchroneous editor), etc.

When a view is updated, a placeholder for a new revision of the full specification is created automatically. This is about the only thing we can expect to be performed automatically and we have to provide facilities to support the process of putting a new revision together. The most general way of working will be to inspect the specification and decide what to integrate.

The support of the merging process is often limited to a syntactical merge of a set of specifications and the result of such a merge may be useful only in some cases (e.g., if all components of the different specifications have got unique identifiers). However, the use of formal specifications opens up for more extensive view integration techniques where for example structural conflicts may be resolved. Something which is very important in a typical modelling situation where different actors in the development process may have different perceptions of the domain (and information system) and thus may lead to different representations. To solve such conflicts a number of conflict resolution techniques are proposed (e.g., [21] for view integration of an extended ER model called ERC+).

Furthermore, we may provide facilities to keep track of changes done to a view after checkout from a full specification. The changes could be recorded textually (i.e., the user get a list of operations and objects upon which the operations are performed) or shown explicitly relative to the components in the corresponding full specification (e.g., shown with a particular notation in a diagram).

### 6.3 Management of Views

**Version Control and Configuration Management** Supporting different views and various versions of the same views demands for a *versioning mechanism* to keep track of the various versions of the same full specification. Versioning of views does not require any special version control factility except from including an appropriate *naming schema* for views. Thus, the version control system to be developed for PPP (Phenomena, Processes, and Programs [15]) and TEMPORA specifications (see [31]) can be applied.

**Impact on the Specification Database** The abstractions as presented here will not require any extensions to the existing class structure in TEMPORA, that is, the existing classes represented in PROBE [7] can still be used [31]. The views that result from applying abstractions are simplifications of the TEMPORA conceptual languages and no additional concepts are added. This implies that the existing structure can be reused, however, certain parameters will become superfluous.

**Efficiency Issues** The efficiency issues of concern are mainly how many views per full specifications that should be kept and how efficient these can be stored. At this stage it seems to be reasonable to say that views which can be derived from the full specification should be derived and not stored, whereas the others (such as restructured abstractions) should be stored. Efficiency issues concerning the storage of the views are mainly determined by the efficiency of the difference processor (e.g., UNIX diff) and is therefore, taken care of by the versioning system.

### 6.4 Different Applications of Abstractions

Abstractions and corresponding views may be used for a number of purposes, for instance:

- in information gathering throughout the modelling phase
- in information retrieval (e.g., rule retrieval)
- for documentation
- for explanation of parts of specifications (e.g., as a supplement to paraphrazing by visualize views of a specification)
- in validation

In general, providing support for individual views contributes to illuminate more aspects of a domain and hopefully, a better understanding and representation of the problem and target IS result. It may also contribute to obtain consensus about a corporate model by allowing different views to be taken account of in the early phases of development.

## 7 Concluding Remarks and Future Directions

We have presented an abstraction-based rule approach to large-scale information systems development. In particular, we have identified the requirements to the development environment and provided a framework for describing the building blocks of such an approach. Furthermore, we have suggested a number of abstraction mechanisms for the different modelling languages in TEMPORA and outlined how these can be applied to deal with large number of rules in systems development (e.g., how to gather high-level and low-level requirements together). It has also been argued that a prerequisite for an abstraction-based approach is formal specifications and comprehensive tool support.

The abstraction approach suggests a systematical way of doing simplifications to a specification and a way to split and provide different views into the same specification (and accordingly, to the Universe of Discourse). The user may go back and forth between simplified specifications and the corresponding specification having the full complexity. In this way, the abstraction facility makes it possible to deal with details at a level natural to the different actors involved (developers as well as end-users). The power of the abstraction facility is further strengthened by allowing updates of abstractions.

The following issues are to be addressed in future work:

- *Dealing with inconsistencies in views.* Investigate how inconsistencies of full specifications and belonging updated views can be detected and removed in the merging process.
- *Tool support and abstraction process description.* Comprehensive tool support and guidelines/methods for how to apply abstractions in the modelling process.

- *Separation of layout and content.* Investigate how ODA (Office Document Architecture [12]) can be integrated in the TEMPORA approach to separate layout and contents of views.
- *Multi-user support.* Investigate how CSCW techniques can be applied to assist the merging process when a group of developers is involved.

## 8 Acknowledgement

The work reported has partly taken place in the ESPRIT II Project TEMPORA. The project is funded by the Commission of the European Communities under the ESPRIT R&D programme. The partners in the TEMPORA consortium are: BIM (Belgium), Hitec (Greece), Imperial College (UK), Logic Programming Associates (UK), SINTEF (Norway), SISU (Sweden), University of Liege (Belgium) and UMIST (UK).

## References

1. M. Alford: Strengthening the Systems/Software Engineering Interface for Real Time Systems, Ascent Logic Corporation, 1991.
2. Barker et al: Expert Systems for Configuration at Digital: XCON and Beyond, Communications of the ACM, Volume 32, Number 3, March 1989.
3. S. Berdal, S. Carlsen: PIP - Processes Interfaced through Ports, Technical Report, IDT, NTH, 1986.
4. V. Berzins and M. Gray: Analysis and Design in MSG.84: Formalizing functional specifications, IEEE Trans. Softw. Eng. , Aug. 1985.
5. V. Berzins et al: Abstraction-Based Software Development, Communications of the ACM, Vol. 29, No. 5, May 1986.
6. V. Berzins and Luqi: Languages for Specification, Design, and Prototyping, In P. A. Ng and R. T. Yeh (Eds.): Modern Software Engineering Foundations and Current Perspectives, Van Nostrand Reinhold, New York, 1990.
7. BIM: BIM_PROBE Manual, BIM, Belgium, June 1990.
8. J. A. Bubenko: Problems and Unclear Issues with Hierarchical Business Activity and Data Flow Modelling, SYSLAB Working Paper no. 134, Stockholm, 1988.
9. C. J. Date: An Introduction to Database Systems, Addison-Wesley Publishing Company Inc., 1986.
10. C. Gane and T. Sarson: Structured Systems Analysis: tools and techniques, Prentice-Hall, 1979.
11. S. McGinnes: How Objective is Object-Oriented Analysis?, In P. Loucopoulos (Ed.): Advanced Information Systems Engineering, 4th International Conference CAiSE'92, Manchester, U. K., 1992.
12. ISO/DIS 8613: Information Processing - Text and office systems - Office Document Architecture (ODA) and interchange format, draft version, The International Standardization Organization (ISO), 1986.
13. J. Krogstie, P. McBrien, R. Owens, and A. H. Seltveit: Information Systems Development Using a Combination of Process and Rule Based Approaches,

In R. Andersen, J. A. Bubenko, and A. Sølvberg (Eds.): Advanced Information Systems Engineering, 3rd International Conference CAiSE'91, Trondheim, Norway, 1991.

14. Xiaofeng Li: What's So Bad About Rule-Based Programming, IEEE Software, Vol. 8, No. 5, Sept. 1991.
15. O. I. Lindland et al: PPP - An Integrated CASE Environment, In R. Andersen, J. A. Bubenko, and A. Sølvberg (Eds.): Advanced Information Systems Engineering, 3rd International Conference CAiSE'91, Trondheim, Norway, 1991.
16. Luqi et al: Graphical tool for computer-aided prototyping, Information and Software Technology, Vol. 32, No. 3, April 1990.
17. McBrien, M. Niezette, D. Pantazis, A. H. Seltveit, U. Sundin, B. Theodoulidis, G. Tziallas, and R. Wohed: A Rule Language to Capture and Model Business Policy Specifications, In R. Andersen, J. A. Bubenko, and A. Sølvberg (Eds.): Advanced Information Systems Engineering, 3rd International Conference CAiSE'91, Trondheim, Norway, 1991.
18. McBrien, A. H. Seltveit, and B. Wangler: An Entity-Relationship Model Extended To Describe Historical Information, Proceedings of CISMOD'92, Bangalore, India, 1992.
19. A.L. Opdahl: RAPIER - A Formal Definition of Diagrammatic Systems Specifications, M.Sc. Thesis, Dept. of Electrical Engineering and Computer Science, IDT, NTH, 1988.
20. R.P. Owens: Notes on the TEMPORA Computation Model, E2469/IC/3.4/7/1, December, 1990.
21. C. Parent, S. Spaccapietra: View integration: a step forward in solving structural conflicts, EPFL-Computer Sc. Dept. Lausanne, Research Report, Aug. 1990.
22. U. Persson, A.H. Seltveit, B. Wangler, R. Wohed: Experience from the Sweden Post Case Study, E2469/SISU/T10.1/12/1, Nov. 1991.
23. T. Pettersen, A.H. Seltveit: A Proposal for a Rule Structuring Mechanism in TEMPORA, E2469/SINTEF/NT1.2/2/1, April 1992.
24. A.H. Seltveit: A Proposal for Abstraction Mechanisms in TEMPORA, E2469/SINTEF/NT1.5/1/1, Nov. 1992.
25. G. Sindre: Rules and Processes in TEMPORA, E2469/SINTEF/T1.1/11/1, Oct. 1989.
26. G. Sindre: Hicons: A General Diagrammatic Framework for Hierarchical Modelling, Ph. D. Thesis, Dept. of Electrical Engineering and Computer Science, The Norwegian Institute of Technology, Trondheim, July 1990.
27. A. Solvberg: A Model for Spefication of Phenomena, Properties, and Information Structures, IBM Research Laboratory, San Jose, California, 1977.
28. A. Solvberg: On the Specification of Scenarios in Information System Design, IBM Research Laboratory, San Jose, California, 1977.
29. TEMPORA: Concepts Manual, Sept. 1990.
30. TEMPORA: The Sweden Post Case Study, Nov. 1991.
31. TEMPORA: Concepts Manual, Sept. 1992.
32. D. C. Tsichritzis and F. H. Lochovsky: Data Models, Prentice-hall, Inc. ,New Jersey, 1982.
33. S. Weiss and M. Page-Jones: Synthesis: An Object-Oriented Analysis and Design Method, Macmillan, 1991.
34. R. J. Wirfs-Brock et al: Designing Object-Oriented Software, Prentice-hall, Inc. , 1990.