The M*-object methodology for information system design in CIM environments : the conceptual design phase

Giuseppe Berio, Antonio Di Leva, Piercarlo Giolito, François Vernadat

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The M-object methodology for information system design in CIM environments : the conceptual design phase*

Antonio DI LEVA
Piercarlo GIOLITO - François VERNADAT

N° 1919
Mai 1993

PROGRAMME 5

Traitement du signal,
automatique et
productique

THE M*-OBJECT METHODOLOGY FOR INFORMATION SYSTEM DESIGN IN CIM ENVIRONMENTS: THE CONCEPTUAL DESIGN PHASE

LA METHODOLOGIE M*-OBJECT POUR LA CONCEPTION DES SYSTEMES D'INFORMATION EN ENVIRONNEMENTS DE CIM : LA PHASE D'ANALYSE CONCEPTUELLE

Giuseppe BERIO*, Antonio DI LEVA*, Piercarlo GIOLITO*, François VERNADAT**

* Politecnico di Torino, Dipartimento di Automatica e Informatica, corso Duca degli Abruzzi 24, I-10129 Torino, Italy

** Università di Torino, Dipartimento di Informatica, corso Svizzera 185, I-10149 Torino, Italy

*** INRIA-Lorraine/CESCOM, 4 rue Marconi, F-57070 Metz, France

Résumé
M*-OBJECT est une méthodologie pour l'analyse, la conception et l'implantation des systèmes d'information de CIM. Dans ce rapport, on présente la phase d'analyse conceptuelle. Elle commence à partir des besoins des utilisateurs et fournit une spécification conceptuelle, ou schéma conceptuel, du système d'information requis par le système de production intégrée à concevoir. Cette phase méthodologique s'appuie sur le modèle PDN (Process and Data Net) qui intégre un modèle de données orienté objets, un langage de manipulations de données et de requêtes, un modèle de traitements et un modèle de description de comportement d'objets. Les traits principaux de l'approche de spécification décrite sont : 1) les propriétés statiques, dynamiques et comportementales de l'information sont toutes traitées, 2) les structures de données et les manipulations de données complexes peuvent être spécifiées et 3) les spécifications sont exécutables pour faire du prototypage rapide.

Mots-clés
CIM, Conception de systèmes d'information, Analyse conceptuelle, Modèle de données orienté objects, Modèle PDN, M* methodology

Abstract
M*-OBJECT is a methodology for the analysis, design and implementation of CIM information systems. In this paper, the conceptual design phase is presented. It starts from users information requirements and provides a conceptual specification, the conceptual schema, of the information system to be used in the integrated manufacturing system to be designed. This methodological phase is supported by the Process and Data Net (PDN) model which integrates an object-oriented data model, a query and data manipulation language, a process model, and an object behaviour description model. The major features of the specification approach are: (1) static, dynamic, and behavioural properties of information are fully covered, (2) complex data structures and data manipulation can be specified, and (3) specifications are executable for rapid prototyping.

Keywords
CIM, Information system design, Conceptual design, Object-oriented data model, PDN model, M* methodology

1. Introduction

Information systems are at the heart of Computer-Integrated Manufacturing (CIM) systems [1] and their design is a crucial step conditioning CIM implementation. Our work in this field started with the M* methodology which is devoted to information system analysis and design for CIM environments [2,3]. Since the object-oriented approach [4] is gaining tremendous interest in the CIM community because of its modelling power and expressiveness, the M* methodology has been adapted to this new paradigm, giving birth to the M*-OBJECT methodology.
The M*-OBJECT methodology consists of three main phases: (a) Organisation Analysis, (b) Conceptual Design, and (c) Implementation Design.
The Organisation Analysis phase, concerned with global enterprise modelling and information system requirements definition, has been discussed in a companion paper [5]. Methods and models used in this phase derive from a modelling paradigm based on three modelling concepts:

- hierarchical decomposition of business activities of the object enterprise;
- specification of the intrinsic behaviour of the resulting enterprise components;
- functional description of interactions between the components.

This paper focuses on the Conceptual Design phase, i.e. the detailed specification of the information system using object orientation. It deals with the analysis of the organisational description and the design of a conceptual description (the *conceptual schema*) of the object enterprise which covers data representation and data processing aspects.
The modelling paradigm used in this phase is completely different from the one used in the previous phase. Conceptual Design is typically a bottom-up design phase in which, starting from elementary components of the object enterprise, simple concepts are modelled first, and more complex concepts are then built (at increasing levels of abstraction) and later merged into environment schemata, and finally aggregated in a conceptual schema. The bottom-up strategy has been chosen to take full advantage of the results obtained in the Organisation Analysis phase. Indeed, if elementary components of the enterprise have already been analysed, the bottom-up strategy is the simplest and most effective choice [6]. This approach is well-suited to CIM since manufacturing system design/redesign is usually based on knowledge from existing systems or from previous solutions. Furthermore, CIM information systems must always be designed after the analysis of the global functional behaviour of the object enterprise.
Models used at the conceptual level of the M*-OBJECT methodology are object-oriented in order to meet the following criteria:

- the introduction of the object type concept (or class), which groups objects which share a common structure and have the same behaviour, is required. Complex objects (composed of simple and/or complex objects) and inheritance are supported;
- the possible evolutions of objects during their life cycle can be specified;
- the interaction (concurrency, parallelism, causality) between activities on objects and components of the enterprise at any level of granularity (environments, organisation units, work-centres) can be described.

One of the major advantages of this approach is to result in detailed executable specifications, which can be used for rapid prototyping.
The Implementation Design phase is dependent on the class of the Database Management Systems (DBMSs) used (relational, object-oriented, or network systems) and on the physical system itself. In M*-OBJECT, it is a performance-driven process that maps the conceptual description of the enterprise onto a logical-physical description that can be directly processed by the target DBMS chosen for the implementation.
Information system design is a time-consuming repetitive job. Currently, many prototypes provide automatic support to designers at several levels: (a) graphical data definition [7]; (b) schema definition and visual querying facilities [8]; (c) visual data browsing [9]; (d) integrated environments, like DDEW [10] which extends the definition process to all phases of database design (from user requirements to physical design), and SUPER [11] which adds a view definition and integration tool to a graphical schema definition and manipulation interface. A prototype of a tool for M*-OBJECT (called M*Designer), which supports analysts in the conceptual design phase, has been developed.
Both the implementation design phase and the conceptual tool will be briefly described in this paper.

The paper is organised as follows. Section 2 provides an overview of the Conceptual Design phase. Section 3 presents the integrated model used at the conceptual level, the Process and Data Net (PDN) model, which integrates an object-oriented data model, a query and data manipulation language, a process description model, and an object behaviour description model. Sections 4 to 7 describe the different tasks of the Organisation Analysis phase. Section 8 describes the Implementation Design phase and Section 9 illustrates the conceptual design tool M*Designer. Section 10 states our conclusions.
Two examples are discussed throughout the paper. The first one concerns a car rental company and is used to illustrate the conceptual model. The other one, which is used to illustrate the methodological tasks, has been introduced in the companion paper [5] and deals with automated manufacturing.

2. The Conceptual Design Phase

The Conceptual Design phase produces a detailed design specification (conceptual schema) of the object enterprise from the high-level, user-oriented description (system environment and system requirements) given by the Organisation Analysis phase. The conceptual schema is an executable specification of static, dynamic, and behavioural features of the enterprise information system.
Static information refers to the structure of the data which characterise the enterprise and to the integrity constraints that data must satisfy. Dynamic information refers to the evolution of the applications used by the organisation under analysis (i.e. the operations that must be performed on data and the causal relationships existing among them) while behavioural information refers to the intrinsic behaviour of the data.
From a terminological point of view, the term "schema" will be used to denote a complete description of a particular object enterprise or part of it expressed in terms of a "model". As suggested in [6], when it is necessary to distinguish between the different components of the schema, we will use the term "D-schema" (for Data schema) to describe static features, and the term "F-schema" (for Functional schema) to describe dynamic and behavioural features.
Inputs to the Conceptual Design phase are system requirements collected in the Organisation Analysis phase [5]. We will assume that they are collected on the basis of the organisation structure of the object enterprise, i.e. for each environment[1] to be automated and for each organisation unit[2] we will assume that requirements consist of:

a) a set of forms, which are documents that are used for data acquisition, output or display (e.g. customer order forms, delivery sheets, reports, computer screen layouts, etc.);
b) a set of record formats, which describe data structures used in traditional file systems (e.g. payroll files, bills of materials files, customer files, etc.);
c) a set of organisation nets and a set of life cycle nets of enterprise components, which describe the flow of control of the processes of the organisation unit;
d) a set of verbal descriptions, which result from interviews and written documents. We assume that they are used to precisely describe the processes of each organisation unit.

The focus of requirements of types (a) and (b) concerns data structures and their properties (static requirements), while the focus of requirements of types (c) and (d) concerns the functional description of processes and activities carried out in organisation units (dynamic requirements) and objects behaviour (behavioural requirements).
The conceptual description of requirements related to a given organisation unit will be called a *local schema*. Each local schema will be designed independently and then integrated, with the other local schemata, into a so-called *environment schema*. The *conceptual schema* is then obtained by integrating all environment schemata. It describes the overall information system (both data and applications). The overall architecture of the Conceptual Design phase is illustrated in Fig. 1.
In the Conceptual Design phase, static, dynamic and behavioural requirements are analysed and modelled (Static Modelling and Dynamic Modelling tasks). These modelling tasks produce D-schemata and activity specifications that are co-ordinated (in the Local Design task) to construct the local schema of each organisation unit.

---

[1] An *enterprise environment* is a consistent subset of the enterprise as defined by users.

[2] An *organisation unit* is an area of responsibility within an enterprise made of work centres or other organisation units. Work centres are elementary areas of work in the enterprise.

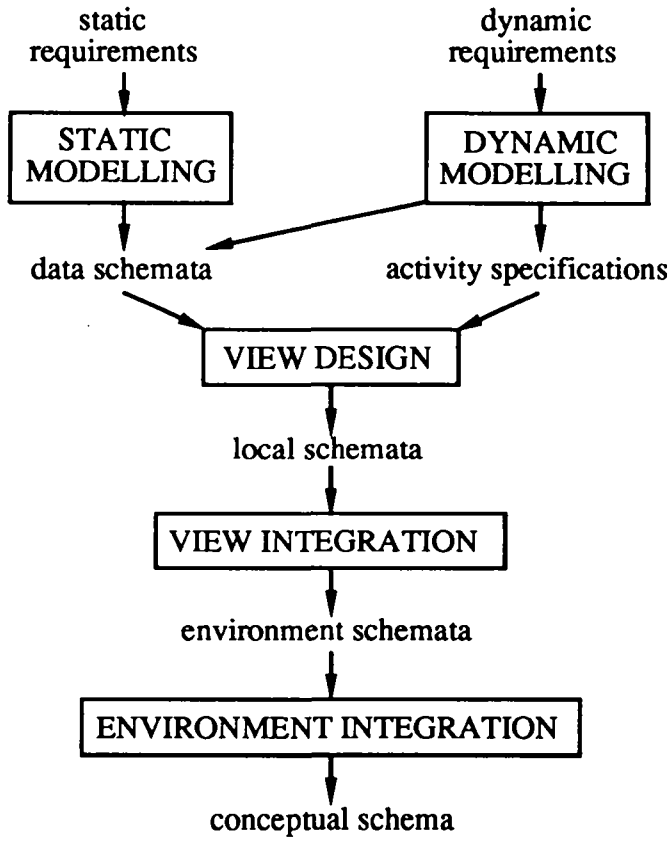


Fig. 1: The Conceptual Design Phase

The local schemata are then integrated to obtain the environment schema (Environment Design task), and a further integration step (Environment Integration task) generates the conceptual schema of the object enterprise.

3. The Integrated Model Supporting Conceptual Design

At the conceptual level, a complete and formal description of all the information aspects of the object enterprise must be produced. These descriptions are specified by means of an executable model, called Process and Data Net (PDN) model, which integrates data representation and data processing.

3.1 The data model

In the PDN model, a system can be described in terms of its *states* which are specified, at any moment in time, as a set of objects stored in the system database and the condition (active or inactive) of the events to which the system is responsive.
Objects represent real-world entities. They are characterised by their class which defines their list of attributes. Objects are also made of a collection of operations (or *methods*) which share a set of internal variables that are accessible only by the object methods. The collection of methods determines the object behaviour and therefore the way its state can evolve.
The data model provides the designer with four basic abstraction mechanisms:

1) *classes*: A class E groups a set of objects (class occurrences) of interest for the system (tools, parts, machines, ...) which share a common structure and have the same behaviour. Each object has an identity, i.e. it has a unique system-defined identifier (OID);
2) *attributes* : The structure of a class E is described by means of a set of attributes. An attribute x of E is a binary relation which describes a characteristic of the class and has a name and a type. If the

type of x refers to a simple or complex domain, it is called a *structural (simple or aggregate) attribute*. If the type of x is a class F (i.e. it can take as values OIDs of F objects), it describes a relationship between the classes E and F and there exists an attribute y of F which describes the inverse relationship; the attributes x and y are called *relationship attributes* and the pair <x, y> will be used to denote the relationship. The properties of an attribute x can be specified by means of two natural numbers (min, max) which mean that an object e of E must take a minimum of min and a maximum of max values. If x is a relationship attribute this means that e must be related to a minimum of min and a maximum of max objects in F, at all times. If min=0 the attribute is optional, i.e. it may not be specified, and a null value (**nil**) is added to each domain or class of the schema to deal with this case. If min≥1 the attribute is mandatory and (at least) a value must be specified for each object. If max=1, the attribute is simple (mono-valued), otherwise it is called multi-valued. A simple attribute can also be calculated. A class identifier of E is a set of one or more attributes which uniquely identify a class occurrence (if this set includes an identifier of another class related to E, E is said to be a dependent class);

3) *data abstraction*: It is the ability to allow the definition of a set of operators (called *methods*) that can be applied to objects of a particular class. Every access to an object is constrained to be via one of the methods. The definition of a class is splitted into two parts, a public *interface*, where structural and relationship attributes are specified and methods are declared, and a private *implementation* part, where procedures and functions are introduced to specify methods and calculated attributes at run-time, respectively. The implementation part may be changed without affecting the interface (*encapsulation principle*);
4) *class hierarchies*: Classes can be organised into class hierarchies. Two types of hierarchies can be used to specify the data structure of the system:

   • a **generalisation** allows a class E (the superclass) to be defined as a generalisation of a set $\{E_1, E_2, \ldots, E_n\}$ of classes (the subclasses or specialisations of E) if each occurrence of a subclass is also an occurrence of the superclass. Every specialisation inherits attributes and methods from the superclass (but not vice versa!) and in addition may have specific attributes and methods. Since generalisation can be applied recursively, classes and superclasses can be organised into a *generalisation hierarchy*. *Two* basic types of generalisation can be considered:

   - the **subset** is the case in which specialisations are not disjoint (i.e. the occurrences of E_1, E_2, ..., E_n are potentially overlapping subsets);
   - the **partition** is the case in which specialisations are disjoint and complete (i.e. each occurrence of E is also an occurrence of one and only one of E_1, E_2, ..., E_n);

   In the PDN model, a generalisation hierarchy can be organised as a direct acyclic graph with one root only, i.e. a class can be a specialisation of several superclasses, but not at the top level! In this way, a set of classes structured into a generalisation hierarchy can be considered as a *hierarchical class* in which the root class is specialised at different level of detail (see the example of Fig. 2);

   • an **aggregation** allows a class (the composite class) to be specified as a composition of existing classes (the component classes). Applying aggregation recursively, an *aggregation hierarchy* can be built. Aggregation hierarchies can be structured as a direct acyclic graph.

A graphical formalism is used to represent such concepts. Rectangular boxes represent classes, circular boxes represent attributes, diamond boxes are used for relationship attributes, dotted arcs denote class identifiers, and different types of arrows represent hierarchies, as described in Table 1.
Figure 2 illustrates a hierarchical class in which a person can be specialised to be an employee or a student or a professor (a student cannot be a professor and vice versa). A subset of students is formed by graduate students, and some of these students can be employed as assistants.
Let us consider now a simple real-world example (adapted from [12]).

The Car Hire (CH) Service : A customer willing to rent a car asks for a car to the service administration of a car rental company. Based on indexes of available and of borrowed cars, a hiring contract is signed. The customer will have a car at his/her disposal and will return it to the service settling the account.

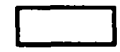
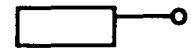
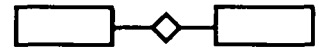
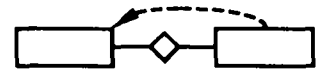
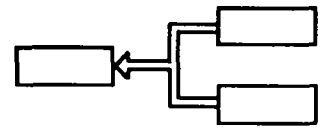
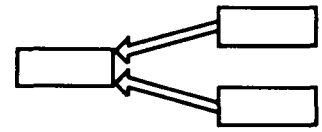
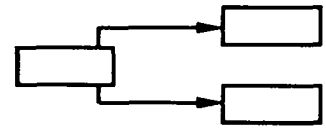
Concepts	Graphical Representation
class	
structural attribute	
relationship attribute	
relationship between classes	
internal identifier	
external identifier	
partition hierarchy	
subset hierarchy	
aggregation hierarchy	

Table 1: Graphical Data Symbols Used in the PDN Model

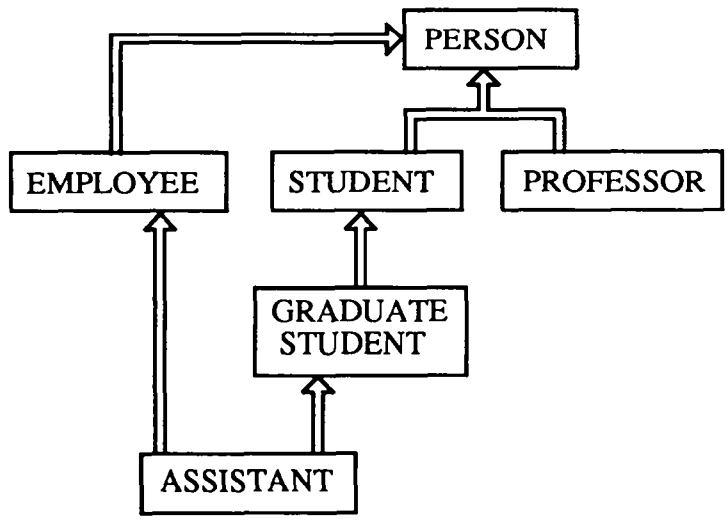
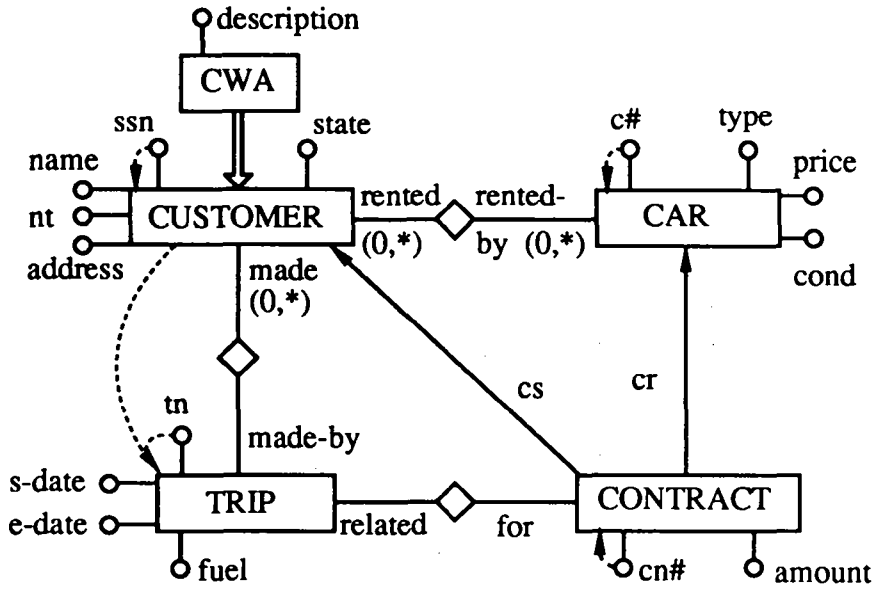


Fig. 2: A Hierarchical Class

In the car rental service database, data about customers, contracts, cars, accounts, and trips performed by customers are recorded. The D-schema for the CH service and its specification are reported in Fig. 3. The schema can be interpreted in the following way:

- CUSTOMERs are identified by their "ssn" (social security number) and have a "name", an "address", and a "state". They "rented" some CARs (in the past) with which they "made" TRIPs, and "nt" is the number of trips made with this service. A customer can be in different states: 'ask': asking, 'ready': ready to use a car, and 'back': returning a car. After the car has been returned, he becomes a 'former' customer. Customers may have an agreement with the car rental service (CWA for customer with agreement), and a "description" describing the agreement conditions;
- a TRIP has a "s-date" and a "e-date" (start and end dates) and a "fuel" consumption. It is "made-by" a customer and "tn" (trip number) identifies the n-th trip made by a given customer. A trip is "related" to a given CONTRACT;
- a CAR is described by means of a "type", a "price" and a condition "cond", which can take one of the values ('free', 'reserved', 'used'). It is identified by a car number "c#" and has been "rented-by" some customers;
- a CONTRACT has an "amount", is identified by a "cn#", is "for" a trip, and is composed of (i.e. it is an aggregation of) a car ("cr") and a customer ("cs").

In the class CONTRACT the attribute "cn#" is calculated. This means that in the implementation part a function cn# must be supplied; in our case, it contains (in its body) the standard function **count** which gives the cardinality of the object domain. Since "cn#" is the identifier, in this case the function cn# gives the cardinality of the class CONTRACT.



class CAR **identifier** c#
member attributes c#, price: **integer**; type: **string**; cond: ('free', 'reserved', 'used');
rented-by (0,*): CUSTOMER **inverse of** rented;
class CUSTOMER **identifier** ssn
member attributes ssn, nt: **integer**; name, address: **string**; state: ('ask', 'ready', 'return', 'former');
rented (0,*): CAR **inverse of** rented-by; made (0,*): TRIP **inverse of** made-by;
subset CWA **of** CUSTOMER **member attributes** description: **string**;
class CONTRACT **composed of** (cr: CAR; cs: CUSTOMER) **identifier** cn#
member attributes cn# (**calc**), amount: **integer** ; for: TRIP **inverse of** related
class CONTRACT **implementation function** cn# : **integer** ; **begin return count** (cn#)+1 **end**
class TRIP **identifier** (made-by, tn)
member attributes tn, fuel: **integer**; s-date, e-date: **date**; related: CONTRACT **inverse of** for

Fig. 3: The Data Schema and the Specification for the CH Service

Objects are manipulated by means of *variables*. A variable x on a class (or a hierarchical class) C takes as value the OID of an object of C, and can be declared by a declaration statement: **use** x|C.
Manipulations on objects are expressed by means of methods of the class they belong to. Methods can be invoked by sending a *request*, with the method name and, may be, some variables, to an object of that class (in the object-oriented literature, a request is usually called a message). The method invocation will be syntactically expressed as:

«variable».«method-name» («input/output-variable-list»)

Methods of a class cannot be invoked in an arbitrary order. For instance, an object cannot be deleted if it was not created first! Therefore, it is useful to group methods related to a given class C into the so-called Object Behaviour (OB) net of that class. The OB net of C describes, for any object $c \in C$, its states, its methods, and the possible execution sequences of method invocations which can be applied to the object. Thus, OB nets represent the conceptual tool to describe the life cycle of an object and will be illustrated in the following.
Methods of a class can be built using pre-defined (or standard) procedures which may return an object that will be bound to a variable of that class. Standard procedures allow to select and manipulate objects (**Create, Update, Delete, Select, Search**), to specify objects in a generalisation hierarchy (**Specialise, Generalise**), and to add (remove) objects into (from) a composite object (**Add, Remove**). For instance, referring to the example of Fig. 3, one can write:

use v|CUSTOMER; x|CAR; y|TRIP; z|CONTRACT
v.**Create** (ssn: 14, name: 'Smith',...); ... «create the customer v whose ssn is 14....»
v.**Specialise** (CWA; description:'...'); ... «customer 14 is specialised as costumer with agreement»
x.**Select** (c#=101); ... «select the car whose number is 101»
y.**Create** (tn=2, made-by: v, ...); ... «create the second trip of the customer v»
z.**Create** (cs:v, ca:x; for: y, ...); ... «create the contract on v and x for the trip y»

3.2 The Query and Data Manipulation Language

The PDN model allows users to express complex queries by means of a well-known mechanism which has been extensively investigated in the context of the relational model, the **view** mechanism [13]. In the relational model, a view is a named "virtual" relation. In the PDN model, a view is a named virtual class, which has the following structure:

view «view-name» («target-list») **use** «variable-declaration-list» **where** «predicate»

where:

- the target list has the following form: $p_h|E_h$, $p_k|E_k$, ... and introduces a set of communication variables p_h, p_k ... on the classes E_h, E_k, ... It specifies what must be retrieved as a set of tuples $<e_h, e_k, \ldots>$ where $e_h \in E_h$, $e_k \in E_k$, and so on;
- in the **use** clause, the variable declaration list has the form $v_m|E_m$, $v_n|E_n$, ... which can be used to introduce a set of local variables on the classes E_m, E_n, ... that must be used to construct the query result;
- the «predicate» in the **where** clause is called a *selective path* because it is, in general, a conjunction of *restriction predicates* (which are used to select relevant objects) and a *path predicate* which specifies a navigation path in the database and assumes the form:
  $p_h<r_{hk}>p_k< \ldots>v_m<r_{mn}>v_n \ldots$
  where r_{hk} is the name of a link which connects E_h with E_k, and so on; a link can be a relationship attribute or an arc of a hierarchy (the name can be omitted if there is just one link between the classes);
- the path predicate is satisfied with the substitution $\{p_h/e_h, p_k/e_k, \ldots, v_m/e_m, v_n/e_n\}$ if, and only if, the object $e_h \in E_h$ is connected by the link r_{hk} to the object $e_k \in E_k$, and so on. A tuple $<e_h, e_k, \ldots, e_m, e_n>$ which satisfies the selective path is called an *occurrence* of the path;

- view occurrences are obtained by discarding from the path occurrences all the variables that are not specified as communication variables;
- the set of occurrences of a view is called the *view extension.*

For instance, referring to Fig. 3, the query:

W "Retrieve trips made by the car with car number 27 in the year 1990"

can be expressed as:

view W (t:TRIP) **use** c:CAR; s:CUSTOMER
where c.c#=27 **and** c<rented-by>s<made>t **and** t.s-date.year≤1990≤t.e-date.year

Manipulations on data are expressed by means of another well-known mechanism in database technology, the transaction mechanism [13], which is called **routine** in the PDN model. A routine describes a state transition, i.e. it both specifies a change in the database state and the event structure of the real-world in which the system is embedded (and to which it is sensitive).
An event is described by means of a *message* which is activated at a given moment in time, may have some variables, and can be declared as:

message «message-name» («variable-list»)

where the message variables can be used to exchange data between the routine and the external world or the rest of the system.
Changes in the database state are represented as input and output views which describe changing data. The change in the event structure is represented by means of input and output messages.
A routine has the following general form (in which a place can be a message or a view):

routine «routine-name» [**accept** «input-place-list»; **return** «output-place-list»]
use «local-variable-declaration-list» ; «local-view-list»
guard «predicate»
exec «routine-block» **end**

where:

- a place is specified by a term of the form $P(p_1, ..., p_n)$ where P is the place name and $p_1, ..., p_n$ is a list of variables corresponding to the formal variables in the place declaration (variables refer to values if P is a message, to database objects if P is a view);
- an input message describes an event that must be active for the routine execution, while an output message describes an event that becomes active after the routine execution;
- input views (in the **accept** clause) describe the data that will be updated by the routine, while output views (in the **return** clause) describe the data that have been updated. A variable which appears in the input and output list means that the *same* object which is received in input is also given back (some of its properties can be changed by the routine);
- with the **use** clause it is possible to introduce local variables (as in the view definition) and local schemata which describe database objects that will not be changed;
- the «predicate» in the **guard** clause can be used to specify conditions on the input and local data by means of path and restriction predicates;
- the «routine-block» consists of a sequence of operators which manipulate database objects; control structures (including *sequence*, *conditional*, and *iterative* structures) can be used to specify the program flow.

For instance, the state transition:

Find: "Find a car of a given type for a given customer and sign the related hire contract"

can be described in the following way:

message INC (ssn: **integer**, type: **string**, dt: **date**)
message CNT (number: **integer**)
view CASK (c:CUSTOMER) **where** c.state='ask'
view CRDY (c:CUSTOMER) **where** c.state='ready'
view FREC (v:CAR) **where** v.cond='free'
view RSVC (v:CAR) **where** v.cond='reserved
view TRCN (t:TRIP, s:CONTRACT) **where** s<for>t
routine Find [**accept** INC(sn, tp, dt),CASK(c),FREC(v) **return** CNT(cnt), CRDY(c), RSVC(v), TRCN(s, t)]
guard c.ssn=sn **and** v.type=tp
exec s.**Create** (cs:c, cr:v); t.**Create** (tn: c.nt+1, s-date: dt, made-by: c, related: s); s.**Update** (for: t);
c.**Update** (state:'ready', nt: nt+1, made: made+t); v.**Update** (cond:'reserved'); cnt:= s.cn# **end** Find

where messages INC and CNT represent a customer asking for a car of a given type and the contract number, respectively. Views CASK and CRDY represent customers willing to hire a car and ready to use it, FREC and RSVC are free and reserved cars, and TRCN contains new contracts and the related trips.

3.3 The process description

In the PDN model, the control policy of a part of the object system (or a *process*) is specified by means of a process net, i.e. a set of routines and causal relationships between them. A process net consists of the following constituents:

1) a *directed net* (**P**, **T** ; **F**) that expresses the control structure of the system. **P** is a set of *places*, **T** is a set of *routines* on the database, and **F** is a set of *arcs* which connect each routine T to its input and output places, i.e.:

   $\mathbf{P} \cap \mathbf{T} = \emptyset$, $\mathbf{P} \cup \mathbf{T} \neq \emptyset$, $\mathbf{F} \subseteq \mathbf{P} \otimes \mathbf{T} \cup \mathbf{T} \otimes \mathbf{P}$[3]

   The set of places is composed of two disjoint subsets: $\mathbf{P} = \mathbf{V} \cup \mathbf{M}$, where **V** is a set of *views* on the database, and **M** is a set of *messages*;
2) a database schema which specifies system objects;
3) a marking of the process net, which consists of the collection of the extensions of its views and the condition (active or inactive) of its messages. An *input substitution* for a routine is a substitution on its input and local views, i.e. an assignment of objects to view variables;
4) a *transition rule* which specifies the system evolution, i.e. how the system reacts to a given marking. For the PDN model, the transition rule describes the enabling and the execution of each routine T of a process net.

   A marking *enables* T if all the input messages of T are active, all the output messages of T are inactive, and there exists an input substitution on the variables of the input and local schemata which satisfies the **guard** predicate.

   If the routine T is enabled it can be *executed*, i.e. statements of the **exec** clause are performed on the objects extracted with the input substitution. The execution must be such that input objects are removed from the input views, output objects are added to the output views, input messages become inactive (they no longer hold) and output messages become active (they begin to hold).

For instance, the process net which models the dynamics of the CH service is illustrated in Fig. 4 (messages are represented by triangles, views by circles, and routines by bars).
The process net of the CH service can be interpreted in the following way:

- a Test has to be done on an incoming person (asking for a car of type 'tp' at a given date 'dt') to check if that person is a former customers; if this is true, data are retrieved (routine Rtrc) from the database, otherwise data describing the new customer must be supplied (message CSD) and inserted (routine Insc) in the database;
- a car of the required type must be selected from the set of free cars, the contract and the related trip must be signed and inserted into the database (routine Find);
- the customer uses the car and then returns the car to the service (routine Usec) ;
- at the end of the trip, the car related to the contract 'cnn' (given by the message END) is controlled, the customer settles the account for the trip, and the contract and the trip are updated (routine Stcn).

---

[3] $\otimes$ represents the Cartesian product of two sets.

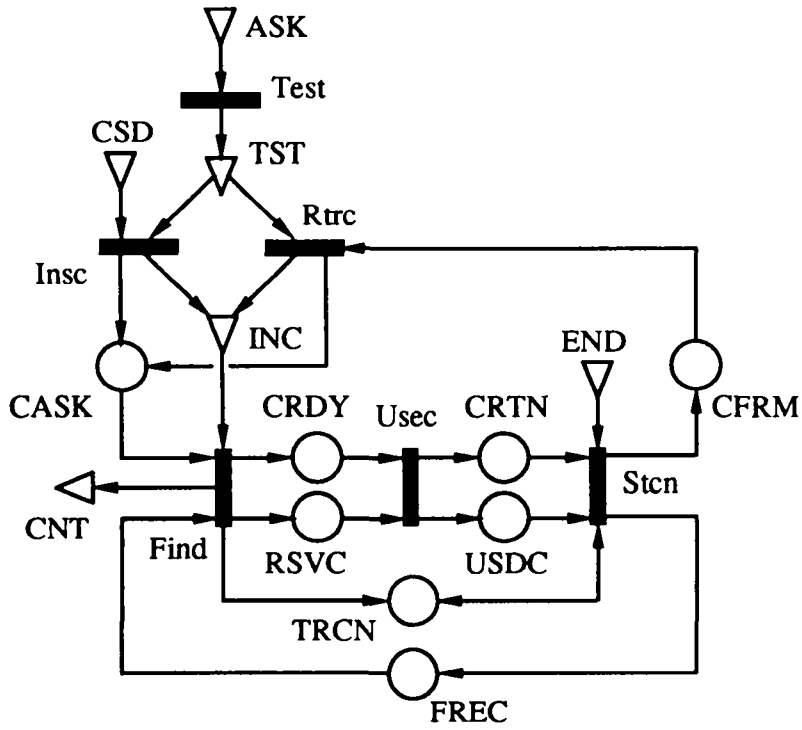


Fig. 4: The Process Net for the CH Service

The message ASK represents a generic person asking for a car, the message TST contains the data after the person data have been tested, the message CSD has to contain further data concerning a person which must be inserted into the database, and the message END represents the end of the trip and the storage of trip data into the database. Views CRTN and CFRM represent customers returning a car and former customers, and USDC contains data about cars used.
Figure 5 shows the specification of the process net in the PDN model.

```
message ASK (ssn: integer; type: string; dt: date)
message TST (ssn: integer; result, type: string; dt: date)
message INC (ssn: integer; type: string; dt: date)
message CNT (number: integer)
message CSD (name, address: string)
message END (cnn, fuel, bill: integer; dt: date)
view CASK (c|CUSTOMER) where c.state='ask'
view CRDY (c|CUSTOMER) where c.state='ready'
view CRTN (c|CUSTOMER) where c.state='back'
view CFRM (c|CUSTOMER) where c.state='former'
view FREC (v|CAR) where v.cond='free'
view RSVC (v|CAR) where v.cond='reserved'
view USDC (v|CAR) where v.cond='used'
view TRCN (t|TRIP, s|CONTRACT) where s<for>t
routine Test [accept ASK(sn, tp, dt) return TST(sn, res, tp, dt)] use c|CUSTOMER
    exec c.Select (ssn=sn); if c=nil then res:='new' else res:='old' end Test
routine Insc [accept TST(sn, res, tp, dt),CSD(cname, caddress) return INC(sn, tp), CASK(c)] guard res='new'
    exec c.Create (ssn:sn, name:cname, address:caddress, nt:0, state:'ask') end Insc
routine Rtrc [accept TST(sn, res, tp, dt), CFRM(c) return INC(sn, tp, dt), CASK(c)] guard c.ssn=sn and res='old'
    exec c.Update (state:'ask') end Rtrc
routine Find [accept INC(sn, tp, dt), CASK(c), FREC(v) return CNT(cnt), CRDY(c), RSVC(v), TRCN(s, t)]
    guard c.ssn=sn and v.type=tp
    exec s.Create (cs:c, cr:v); t.Create (tn:c.nt+1, s-date:dt, made-by:c, related:s); s.Update (for:t);
    c.Update (state:'ready', nt:nt+1, made:made+t); v.Update (cond:'reserved'); cnt:= s.cn# end Find
routine Usec [accept CRDY(c), RSVC(v) return CRTN(c), USDC(v)] use x|CONTRACT guard c<cs>x<cr>v
    exec c.Update (state:'return'); v.Update (cond:'used') end Usec
routine Stcn [accept END(cnn, fl, bl, dt), CRTN(c), USDC(v), TRCN(s, t)
    return CFRM(c), FREC(v), TRCN(s, t)]
    guard s.cn#=cnn and s<for>t and c<cs>s<cr>v
    exec c.Update(state:'former'); v.Update(cond:'free'); s.Update(amount:bl, e-date:dt); t.Update(fuel:fl) end Stcn
```

Fig. 5: The Specification of the Process Net for the CH Service

3.4 The object behaviour description

A basic feature of the M*-OBJECT methodology is the distinction between system capabilities, i.e. what the system can do (behavioural features) and the control strategy which includes the logic of system evolution, i.e. what has to be done (dynamic features). The suggested policy consists in describing the overall behaviour of an application (what has to be done) by means of one or several process nets, and the intrinsic behaviour of the objects described in the database schema (what the system can do) by means of Object Behaviour (OB) nets.
By intrinsic behaviour, we mean a set of operations and request protocols representing a program component common to different applications of the information system. Usually, this is related to the objects of a particular class which describes the component, and to the life cycle of the component brought into focus in the Organisation Analysis phase.
OB nets belong to a particular class of nets in which the following interpretation has been adopted:

1) an object behaviour net of a class C consists of a directed net (**S**, **M** ; **F**) that expresses the intrinsic behaviour of its objects. **S** is a set of views which represent different states of objects c of C, **M** is the set of methods which transform an object state into another object state, and **F** is a set of arcs which connects each method to its input and output places. A path in the object behaviour net determines a possible sequence in which methods of C can be executed (an object behaviour net is then equivalent to a finite state automaton);
2) a method M is *callable*, i.e. it can be invoked, for an object c∈ C if, and only if, c belongs to its input view; when the method is executed the object changes its state (i.e. it "flows" to the output view);
3) unlike PDNs (where a routine can be executed when it is enabled) a method in an object behaviour net must be invoked by a routine T of a related process net to be executed. T can invoke M for an object c of C only if M is callable for c.

A method specification has the following general form:

method «method-name» («input/output-variable-list»)
accept «input-view» **return** «output-view»
use «local-variable-declaration-list» **var** «type-variable-declaration-list»
exec «method-block» **end**

where:

- variables in the input/output variable list refer to values or objects exchanged with the environment;
- input and output views describe input and output states of the objects;
- the **use** and **var** clauses have the same meaning illustrated for the routine definition;
- the «method-block» contains a sequence of operations which describes the state modification.

In the CH service organisation, objects with a relevant life cycle are customers and cars (contracts and trips are simply created and stored into the database). The intrinsic behaviour of these objects can be expressed by means of the following set of methods:

Customers

ins-cs : «insert a customer»
cs-ar : «customer becomes ready»
cs-rb : «customer returns the car»
cs-bf : «customer gets the "former" state»
cs-fa : «a known customer asks for a car»

Cars

cr-fr : «reserve a free car»
cr-ru : «use a reserved car»
cr-uf : «a car becomes free»

OB nets for the two classes CUSTOMER and CAR are shown in Fig. 6. Object states are represented by means of the same views introduced for the process net of Fig. 4 (but this is not a general rule!).
Using the methods defined in the OB nets, the process net for the CH service assumes the simpler (and more intuitive) form of Fig. 7.

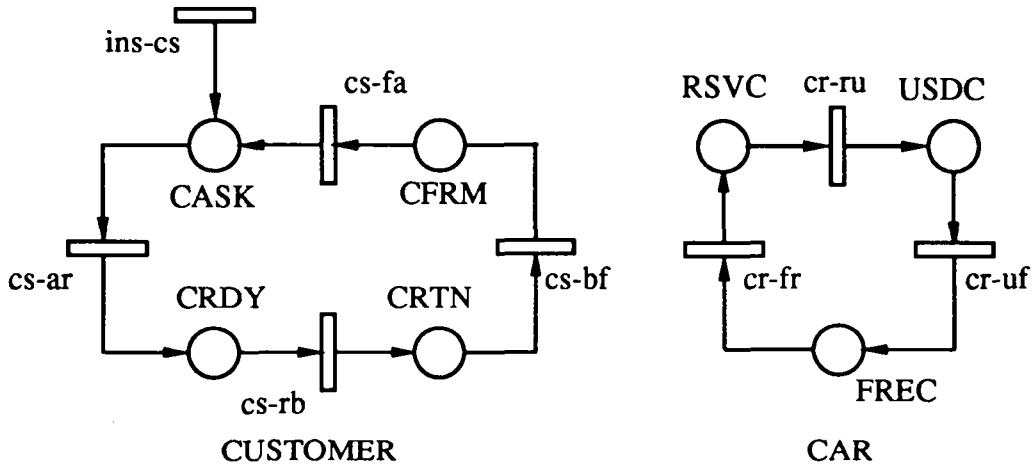
As can be noted comparing Fig. 6 and Fig. 4, the object behaviour nets of Fig. 6 are integrated into the net which expresses the global behaviour of the whole system. This fact outlines a methodological principle that will be illustrated in the following sections.

4. The Static Modelling Task

In this task, all kinds of formatted requirements are considered. Formatted requirements refer to structured descriptions such as:

- Forms exchanged between components of the object organisation and the external world (e.g. customer order forms, delivery sheets, etc.);
- Formatted screen layouts to acquire data and to show program results on a computer terminal;
- Record formats from data files (e.g. payroll file, bill of materials files, etc.);
- Questionnaires used to collect information;
- Data Division from COBOL programs or other similar languages;
- Data schemata defined with data description languages of pre-existing databases.

In this paper, forms and Data Division from COBOL programs will be considered, but the related methodological steps can be easily extended to the other cases.




```
class CUSTOMER ...                    methods ins-cs (...); cs-ar(...); cs-rb; cs-bf(...); cs-fa
class CUSTOMER implementation view CASK(c)...; view CRTN(c)...; view CFRM(c)...; view CRDY(c)...
  method ins-cs (c-ssn: integer; c-name, c-address: string) return CASK(c)
    exec c.Create (ssn: c-ssn, name: c-name, address: c-address, nt: 0, state:'ask') end
  method cs-ar(t:TRIP; s|CONTRACT) accepts CASK(c) return CRDY(c)
    exec s.Create (cs: c, cr: v); t.Create (tn: c.nt+1, s-date: dt, made-by: c, related: s);
      s.Update (for: t); c.Update (state:'ready', nt: nt+1, made: made+t) end
  method cs-rb accepts CRDY(c) return CRTN(c) exec c.Update (state:'back') end
  method cs-bf(fl, bl:integer, dt:date; t:TRIP; s|CONTRACT) accepts CRTN(c) return CFRM(c)
    exec c.Update (state:'former'); s.Update (amount: bl, e-date: dt); t.Update (fuel: fl) end
  method cs-fa accepts CFRM(c) return CASK(c) exec c.Update (state:'ask') end

class CAR ...                         methods cr-fr; cr-ru; cr-uf
class CAR implementation  view RSVC(v)... ; view USDC(v)... ; view FREC(v)...
  method cr-fr accepts FREC(v) return RSVC(v) exec v.Update (cond:'reserved') end
  method cr-ru accepts RSVC(v) return USDC(v) exec v.Update (cond:'used') end
  method cr-uf accepts USDC(v) return FREC(v) exec v.Update (cond:'free') end
```

Fig. 6: The Object Behaviour Nets for the CH Service

```
routine Test [accept ASK(sn, tp, dt) return TST(sn, res, tp, dt)] use c|CUSTOMER
  exec c.Select (ssn=sn); if c=nil then res:='new' else res:='old' end Test
routine Insc [accept TST(sn, res, tp, dt), CSD(cname, caddress) return INC(sn, tp, dt), CASK(c)]
  guard res='new' exec c.ins-cs (sn, cname, caddress) end Insc
routine Rtrc [accept TST(sn, res, tp, dt), CFRM(c) return INC(sn, tp, dt), CASK(c)] guard c.ssn=sn and res='old'
  exec c.cs-fa end Rtrc
routine Find [accept INC(sn, tp, dt), CASK(c), FREC(v) return CNT(cnt), CRDY(c), RSVC(v), TRCN(s, t)]
  guard c.ssn=sn and v.type=tp exec c.cs-ar (s,t); v.cr-fr; cnt:= s.cn# end Find
routine Usec [accept CRDY(c), RSVC(v) return CRTN(c), USDC(v)] use x|CONTRACT guard c<cs>x<cr>v
  exec c.cs-rb; v.cr-ru end Usec
routine Stcn [accept END(cnn, fl, bl, dt), CRTN(c), USDC(v), TRCN(s, t)
  return CFRM(c), FREC(v), TRCN(s, t)]
  guard s.cn#=cnn and s<for>t and c<cs>s<cr>v  exec c.cs-bf(fl, bl, dt, s, t); v.cr-uf end Stcn
```

Fig. 7: Revised Specification of the Process Net for the CH Service

4.1 Form Analysis

Forms are formatted documents that are used within the organisation to manage communications (as input to, as output from, and between functions). Since data are well-structured in forms, the design process is easy and well-defined compared to the design process for non formatted requirements.
A complete methodology to transform form descriptions into D-schemata has been illustrated in [14]. We only summarise here the basic steps of this methodology for the sake of completeness.
The form model assumes that any form can be divided into *areas*. An area is a portion of the form which contains information pertaining to the same argument or describing the same concept. Areas may be further structured into subareas to any level of subdivision, and are composed of *parts* which usually belong to one of the four following types:

1. Certificative parts, which contain information elements (such as signatures, dates, marks, and so on) that certify the correctness of the form and are usually not relevant to the data design process;
2. Extensional parts, which consist of fields that must be filled in by users when the form is completed;
3. Intensional parts, which refer to names describing (in implicit or explicit way) the meaning of values to be put in the extensional parts;
4. Descriptive parts, which specify instructions or rules to be followed when filling in the extensional parts.

Problems with forms come from the great variety of different layouts they can present. Furthermore, semantic information concerning the structure and the meaning of a form can be explicitly expressed or can be contained in undocumented, implicit conventions used in the enterprise.
Usually extensional and intensional parts are mixed and interconnected in a form. A classification of information that appears in extensional/intensional parts of a form depends upon the type of linguistic representation used:

a) Parametric text: It is a text written in natural language with fields that have to be filled with values extracted from suitable domains (e.g.: We certify that ....., born in ..... on .../.../.....). Once such fields have been completed, the text appears as a set of coherent sentences in natural language. Frequently, in such cases, the intensional information (name, city, date_of_birth,...) is not explicitly present and must be elucidated during the design process;
b) Table: It is a bi-dimensional structure in which values of a given property (e.g. a temperature) are represented in terms of two indexes (e.g. month: [1..12] and day: [1..31]) which identify each position in the table. N-dimensional tables with n>2 can be expressed by means of sequences of bi-dimensional tables;
c) List: It is a sequence of all the possible values that a property can hold. The user selects one (or some) of them when the form is filled in.

Descriptive parts concern rules and/or constraints on the way forms must be filled in with data. They can be interconnected with extensional/intensional parts or they can be mentioned apart (e.g. footnotes). Sometimes, rules are not indicated at all since they are based on "obvious" conventions (e.g.: Do not fill the field: "Number_of_pregnancies", if the person is a male), and must be pointed out.

Usually, rules are translated into constraints which can be expressed in the form D-schema or must be enforced within application programs.
Starting from the forms collected in the Requirements Collection step of the Organisation Analysis phase, a set of D-schemata (one for each form) is produced. The Form Analysis task consists of two steps: (a) requirements analysis, in which a form is segmented into meaningful parts and relevant concepts are extracted, (b) form design, in which concepts are modelled and the form D-schema is constructed. The overall architecture of this task is given by Fig. 8.
During the Requirements Analysis step, forms are analysed to determine form areas and to classify different parts according to the four types introduced previously. Areas may be further broken down into subareas. For each area (or subarea), a name must be provided to express, in the most explicit way, the meaning of the argument or concept it describes.
In Fig. 9, areas and subareas of a Process Plan form (see Fig. 11 of [5]) are enclosed within dashed lines. Data in the first row of the form (version, date,...) are considered as certificative information.

foreach organisation unit:
  foreach form:
  Requirements Analysis
    1. distinguish parts (certificative, descriptive, extensional/intensional)
    2. select areas and subareas, and give them a name
    3. extract elementary concepts (and their properties) and add them to the data glossary
  Form Design
    4. repeat
      4.1 select a concept in the glossary
      4.2 classify the concept
      4.3 insert the concept in the form D-schema
      4.4 update the glossary
      until all the concepts have been processed
    5. restructure the form D-schema

Fig. 8: Form Analysis Steps

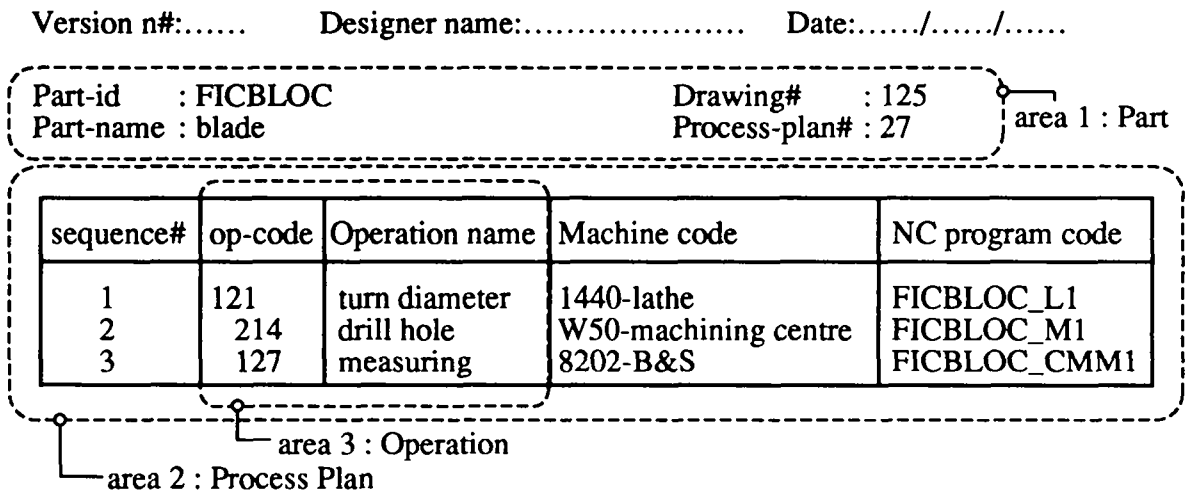


Fig. 9: The Analysis of a Process Plan Form

Then, relevant concepts must be extracted from the object form and inserted into the data glossary. Each entry in the glossary describes a concept which consists of a code, a concept name, a brief description, a domain description, possible instances, synonyms and the abstraction hierarchies in which the concept is involved. Description, synonyms, and hierarchies are used for a better understanding of the meaning of the concept (see Fig. 10).
Concepts are related to names which appear in the intensional and descriptive parts, and to names of areas and subareas. Usually, it is useful to cluster the concepts related to the same area. The specification of concepts depends on the type of linguistic representation (parametric text, table, or list). For parametric text, the names of fields (identifying the elementary concepts) appear implicitly or explicitly within the text. For tables, users have to provide values "indexed" by means of (limited) value sets. The meaning of values and indexing sets usually suggest the elementary concepts that are dealt with. For lists, the name of the related property identifies the concept.
Care must be taken when filling in the glossary. The designer has to solve data synonym, homonym, and duplication problems. Different glossaries must be cross-referenced to verify that concepts referenced in one glossary are presented in the other with the same name [15]. These checks make the methodology quite safe from requirements given by different users.
In the Form Design step, each elementary concept must be classified as a class, a structural or relationship attribute, or as a hierarchy, and inserted into the D-schema, using the glossary as a source of information and cross-reference.
The classification process, which assigns a type to each concept, is complex and difficult to formalise. A concept can be classified as a class if it has properties and it is related to other classes (e.g. it can be considered as a subclass or a component of another class). It can be classified as an attribute if it has no properties, it is natural to find another concept for which the concept is a property, and it is easy to give sample values of it (with their corresponding domains) from the available examples.
The first concepts to be taken into account (step 4.1) must be the most *relevant* concepts referenced in the glossary. Usually, we start from the concepts related to areas and subareas of the form (in most cases, they correspond to classes), and then concepts of the same area are inserted in the D-schema. At this point, the resulting classes must be connected introducing relationship attributes which can be derived from the logical links existing in the real-world. The D-schema is completed by adding cardinalities of the attributes and selecting suitable identifiers for the classes. For instance, Fig. 10 shows the glossary and the resulting D-schema for the form of Fig. 9.
The D-schema of Fig. 10 can be interpreted in the following way:

- PARTs (identified by "part_id") are worked according to their PROCESS-PLAN;
- a PROCESS-PLAN (identified by the number "pp#") consists of a set of PROCESS-STEPs related to the given PROCESS-PLAN by means of the relationship attribute "sp";
- a PROCESS-STEP corresponds to a row in the Process Plan form and it is described by means of a progressive sequence number "s#" (which specifies the order in which operations must be applied) and the OPERATION to which it is related;
- an OPERATION is identified by means of a code "oc" and can be related to several process steps.

The aim of last analysis step (restructuring) is to increase the clarity and the expressiveness of the D-schema by means of data modifications. Since clarity and expressiveness are difficult to quantify, this step is highly informal. Usually, a global analysis of the whole D-schema may suggest some restructuring. For instance, if two classes have many properties in common (while representing different concepts), they can be merged into a unique hierarchy. The introduction of new abstractions is justified when they add further clarity and simplicity to the D-schema. It is discouraged when "unnatural" classes would result from this operation.

4.2 Record Format Analysis

Record formats describe the structure of data used in applications written in COBOL (similar descriptions apply for other procedural programming languages such as C, FORTRAN or PL/1).
Record formats are declared in the DATA DIVISION part of the COBOL programs. Records are collected into *files* and have a hierarchical structure, i.e. they consist of groups of *fields*, which in turn can be composed of subfields, and so on. A field is described giving its level in the hierarchy and its type or PICture (a code X or 9 specifies an alphanumerical or a numerical field respectively). A clause

OCCURS n TIMES (with a given integer n) specifies repetitive fields, which can be nested to represent tables.
Starting from the record formats collected in the Requirements Collection step of the Organisation Analysis phase, a set of D-schemata (one for each record format) is produced. The Record Format Analysis task consists of two steps: (a) requirements analysis, in which record fields are classified and their meaning is recognised, and (b) record format design, in which records and fields are modelled and the record D-schema is constructed. The overall architecture of this task is displayed in Fig. 11.

code	name	description	domain	instance	syn	area
D1	PART	Part to be worked				1
D2	part_id	part identifier	C(10)	FICBLOC		1
D3	p_name	part name	C(20)	blade		1
D4	dr#	drowing number	int(3)	125		1
D5	PROCESS PLAN	process plan				2
D6	pp#	process plan number	int(3)	27		1
D7	PROCESS STEP	step of a process plan				2
D8	s#	sequnce number	int(2)	1,2,3,...		2
D9	mch_cd	machine code	C(10)	1440-lathe		2
D10	NC_cd	NC program code	C(10)	FICBLOC_L1		2
D11	OPERATION	Operation				3
D12	oc	operation code	int(2)	121,214,...		3
D13	op_nm	operation name	C(20)	measuring		3

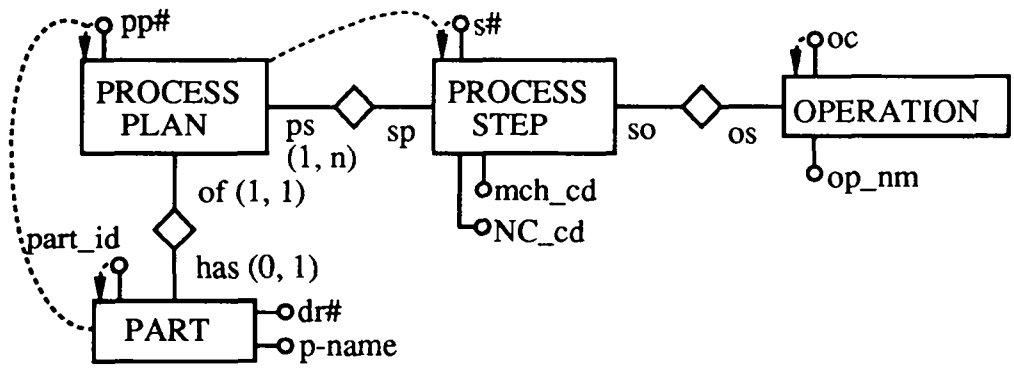


Fig. 10: The Design of the Process Plan Form

foreach organisation unit:
  foreach file described by a record format:
  Requirements Analysis
    1. classify fields
    2. recognise the meaning of each field
  Record Format Design
    3. model records and fields and insert them into the D-schema
    4. specify classes by introducing relationship attributes and hierarchies
    5. restructure the D-schema

Fig. 11: Record Format Analysis Steps

During the Requirements Analysis step, fields must be classified and the meaning of each field must be recognised. Eventually, users can be interviewed or the program which uses the data can be inspected. A field can be classified (according to the type of information it contains) as:

a) Data field: It contains a value, extracted from a suitable domain, which describes a property of the real-world entities represented in the file;
b) Pointer field: It contains a value which can be used to access another record (of the same or another file);
c) Discriminating field: It contains a value used to indicate if the subsequent field(s) can take a value. Sometimes, according to different values, some fields can be enabled or not, or their interpretation change. Discriminating fields can be used to express intricate programming rules that are difficult to understand. In this case, users interviews may help to elucidate the right meaning.

In the Record Format Design steps, a class is used to represent the record format of each file, and simple, repetitive, or compound data fields are translated into simple, multi-valued, or aggregate structural attributes, respectively. Fields which can be used to identify the records (keys) must be recognised and translated into class identifiers. Pointers are used in COBOL to represent relationships between records of different files or of the same file. Thus, they can be translated as relationship attributes. Also aggregation hierarchies can be used in this case (component objects and compound objects are described in different files). Discriminatig fields usually refer to generalisation hierarchies. The D-schema is completed by adding the cardinalities of the attributes. Since there is a semantic gap between COBOL descriptions and the rich set of abstractions that can be used in the M*-OBJECT data model, completion of the D-schema usually requires additional interviews of users on the correct meaning of all the concepts described in the D-schema. The last step (restructuring) aims at increasing the clarity and the expressiveness of the D-schema by means of data modifications.
For instance, the COBOL specification of the data structures of the application program which manages the data for the CH service is illustrated in Fig. 12 while Fig. 13 shows the D-schema for the bill of materials (BOM) record format (see Fig. 12 of [5]).

```
DATA DIVISION.                                              comments
  WORKING_STORAGE SECTION.
  01   CAR.
       02   C#              PIC 9(4).                       key
       02   TYPE            PIC X(20).
       02   PRICE           PIC 9(8).
       02   COND            PIC X.                          possible values: 'F','R','U'
  01   CUSTOMER.
       02   SSN             PIC 9(10).                      key
       02   CAR-USED        PIC 9(4) OCCURS 10 TIMES.       pointers to CAR
       02   NT              PIC 9(3).
       02   NAME            PIC X(10).
       02   ADDRESS         PIC X(20).
       02   STATE           PIC X(2).                       possible values: 'AS','RD','RT','FR'
       02   TYPE            PIC X(2).                       Discriminatig field (possible values: 'A','N')
       02   DESCRIPTION     PIC X(20).                      agreement description (enabled if TYPE='A')
  01   CONTRACT.
       02   CN#             PIC 9(8).                       key
       02   CS              PIC 9(10).                      pointer to CUSTOMER
       02   CR              PIC 9(4).                       pointer to CAR
       02   FOR             PIC 9(2).                       pointer to TRIP
       02   AMOUNT          PIC 9(8).
  01   TRIP.
       02   TN              PIC 9(2).                       key
       02   MADE-BY         PIC 9(10).                      key - pointer to CUSTOMER
       02   S-DATE          PIC X(8).
       02   E-DATE          PIC X(8).
       02   FUEL            PIC 9(3).
```

Fig. 12: COBOL Specification of the Data Schema for the CH Service

```
DATA DIVISION.
WORKING_STORAGE SECTION.
 01 ITEM.
   02 PARTNO        PIC 9(6).
   02 DESC          PIC X(20).
   02 LEAD_TIME     PIC 9(3).
 01 COMPRISES.
   02 LEVEL         PIC 9(2).
   02 ITEM          PIC X(10).
   02 SUBITEM       PIC X(10).
   02 QTY           PIC 9(2).
```

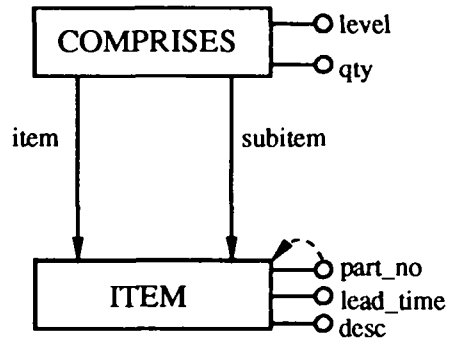


Fig. 13: The BOM Record Format and the Related Data Schema

5. The Dynamic Modelling Task

The aim of the Dynamic Modelling task is to analyse the unformatted description of the functional and operational characteristics concerning those parts of the enterprise which have to be automated. They result from interviews and written descriptions. Such types of requirements are expressed by means of natural language sentences. Usually the focus is placed on the functional description of processes and activities, i.e. they are used to finely describe, along with the organisation nets and life cycle nets, the functions to be automated. Users have been encouraged to describe how organisation processes of the object functions are executed and how data are captured, produced, or processed by the activities which constitute the processes.

As noted in [15], natural language descriptions must be restricted according to suitable conventions. The restriction process (or filtering process) aims at:

- simplifying sentences;
- improving understanding and removing semantic ambiguities of the texts.

These requisites can be acquired, for instance, by:

- reducing repetitions, redundancies, and use of synonyms;
- resolving homonyms by introducing suitable substitutes;
- substituting pronouns with the corresponding nouns and separate complex sentences;
- making explicit elements being implicit;
- applying restrictions to articles, quantifiers, plurals and so on.

A filtered sentence can be classified, according to the type of information it contains, as:

- data sentence: It describes data to be used by an activity of the organisation. It should be of the form: <subject><verb><specification>, where the <verb> part usually refers to structural verbs such as: is a, concerns, has, consists of, contains, includes, is made by, and so on;
- constraint sentence: It expresses a restriction that apply on data, and the <verb> part usually refers to normative verbs such as: must, cannot, has to, and so on;
- activity sentence: It describes an activity of the organisation. It should be of the form: <verb><specification>, where the <verb> part expressing the activity must be in the imperative mode and usually refers to verbs such as: prepare, send, issue, evaluate, generate, and so on;
- event sentence: It describes the condition(s) that triggers an activity. If possible, its structure must be of a form similar to a control structure of a programming language, like:

  execute <activity> before/after <activity> ...
  when <condition> do <activity> ...
  if <condition> then <activity> else <activity> ...

Sentences must then be analysed to check whether completeness and consistency rules are satisfied. The basic rules are the following:

- each datum which is used by an activity sentence must be either described by a data sentence or analysed in the static modelling task (i.e. it belongs to a form or a record format of the same organisation unit);
- each activity which is referred to by an event sentence must be described by an activity sentence;
- each activity must be co-ordinated, at least by one event sentence, to other activities or to temporal conditions (each month, every 10 minutes, at the end of the year,...).

The third step concerns Data Design and Activity Specification. First of all, for each data item referred to in the activity description, a type must be selected (class, attribute, or hierarchy) and then the item is inserted into the activity D-schema. Classes must be logically connected by means of relationship attributes or aggregation arcs, and the activity D-schema must be completed with attribute cardinalities, identifiers, and so on.
The activity is then specified in terms of input and output data, and the manipulations on the database which express the state evolution. More specifically, the activity descriptor specifies:

- input and output requisites (i.e. data and conditions that are necessary to allow the activity to be carried out and data that will be produced);
- what are the life cycles involved in the activity (usually, they refer to "typical" objects which change their state when it is activated);
- how data are transformed.

The overall architecture of the Dynamic Modelling task is given by Fig. 14.

Filtering

1. group sentences related to the same organisation process and filter them

foreach process:

Requirements Analysis

2. analyse and classify sentences (data, constraint, activity, event)
3. verify sentences

Data Design and Activity Specification

4. foreach activity of the object process:
   - 4.1 assign a type to each datum involved
   - 4.2 construct the D-schema
   - 4.3 analyse input and output requisites, and collect relevant life cycles
   - 4.2 construct the activity specification

Fig. 14: Dynamic Modelling Steps

In Fig. 15, the sentences filtered from the description of the Execute process (see Fig. 12 of [5]) have been classified and verified (types 'dt', 'cn', 'ac', 'ev' stand for data, constraint, activity, event, respectively).
As it can be noticed, the requirements have been integrated with new sentences and old sentences have been splitted and modified to specify in a complete and consistent way the process description. In fact, the Requirements Analysis step is usually an iterative process which "refines" user descriptions until no more interpretation problems exist. It must also be noticed that, at this point, the procedural aspects, i.e. *how* operations are performed, must be known.

code	sentence	type
D1	A fabrication lot is a batch of similar parts to be produced	dt
D2	Each occurrence of a part of the same lot is a workpiece made of some material	dt
A1	Manufacture (Work) a part according to a process plan	ac
D3	A process plan is a predefined sequence of machining operations	dt
A2	Mount a part on the appropriate pallet	ac
D4	Parts are processed on a given machine with a given tool	dt
A3	Prepare (Prep) a part for the next operation	ac

A4	Dismount (Dism) a part from a pallet	ac
A5	Send back a part for the next operation	ac
V1	**if** the part is mounted **then** Work	ev
V2	**if** the process plan is completed **then** Dism **else** Send	ev
V3	**when** an operation is terminated **then** Prep	ev

Fig. 15: Filtered Sentences for the Execute Process

In our example, the D-schema and the activity specification corresponding to the sentence A1 (Work) are shown in Fig. 16.

Work	:	sentence A1
input	:	a part (mounted on a pallet) the machine and the tool to execute the current operation on that part are available
output	:	part, machine, and tool are in the working state
life cycles	:	part, machine, tool
operations	:	update the status of part, machine and tool, and connect the given part to the machine and to the tool actually in use

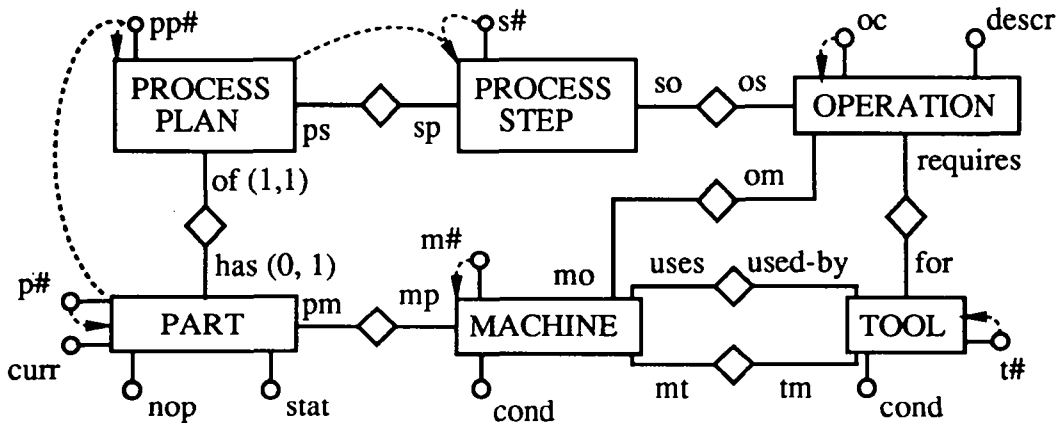


Fig. 16: D-schema and Activity Specification for the Work Activity

In addition to explanations for Fig. 10, the D-schema can be interpreted in the following way:

- PARTs must be processed on a MACHINE with a proper TOOL. Each part is described by means of a "stat"(us) ('ready', 'mounted', 'working', 'worked', 'completed') and a part number "p#";
- to control the manufacturing process, two attributes of PART are used: (1) "nop" which contains the total number of operations to be executed, and (2) "curr" which contains the sequence number of the next step in the process plan to be performed;
- OPERATIONs are described by means of a code "oc" and a "name". Attributes "om" and "requires" relate each operation to the MACHINEs and the TOOL which can be used to perform it;
- MACHINEs are described by means of an identification number "m#" and a "cond"(ition) ('available', 'working'). The attributes "mp" and "mt" specify, respectively, the part which is actually worked by a given machine and the tool which is used to work that part;
- TOOLs are described by means of a tool number "t#" and a "cond"(ition) ('available', 'working'). A tool can be "used-by" several machines for different OPERATIONs.

6. The Local Design Task

In the Local Design task, a local schema is built for each organisation unit of all environments to be automated. Starting from the D-schemata constructed in the Static and Dynamic Modelling tasks, a local D-schema is generated. First of all, form and record format schemata are aggregated to produce a draft D-schema. We start in this way because data appearing in forms and record formats are usually highly structured. Their conceptual analysis is then easier, and the draft D-schema produced is more reliable with respect to the D-schemata produced by the analysis of dynamics requirements (described in a nonformatted way). The aggregation process is then iterated for all the D-schemata of the activities related to the organisation unit being considered.

D-schemata aggregation (or integration) is a critical step because different schemata may be produced by different designers and the same piece of reality can be perceived in different ways [6]. An aggregation step between the draft D-schema and a D-schema to be aggregated is carried out by identifying common concepts, but conflicts may arise because the two D-schemata may have different "views" of the concepts they describe. Such conflicts may involve the name of the concepts (homonyms and synonyms problems) and their type (concepts having the same name but having different representation structures or different properties in the two D-schemata).

For instance, if the D-schemata of Fig. 10, Fig. 13, and Fig. 16 are compared, the following pairs of synonyms can be considered: "name" in Fig. 16 and "op_nm" in Fig. 10, "PART" in Fig. 16 and "ITEM" in Fig. 13, "p#" in Fig. 16 and "part_no" in Fig. 13 or "part_id" in Fig. 10, while "name" specifies the name of an OPERATION in Fig. 16 and the name of a PART in Fig. 10 (homonyms).

The methodology suggests for every case a set of possible solutions that must be evaluated with users, and usually this implies that the D-schemata must be updated to unify the representations (see, for instance, [6] for a detailed discussion of these problems).

When all conflicts have been solved, all the concepts which are common to the two D-schemata have the same name and type, and can be merged. Classes with some attributes different in the two D-schemata are merged by taking the union of their attributes. The aggregated D-schema which includes all the concepts described in the component D-schemata is then obtained by superimposing common concepts.

Restructuring may be applied to increase the clarity and the expressiveness of the D-schema by means of data modifications.

For instance, the D-schema of the Flexible Manufacturing Cell (FMC) organisation unit is illustrated in Fig. 17, for which the following observations hold:

- PARTs must be mounted on a PALLET to be processed by a MACHINE. A pallet is described by means of a "cond"(ition) ('available', 'working') and a pallet number "f#". Parts can be mounted on different pallets (specified by the attribute "pf"), and a pallet can be used for different parts (specified by the attribute "fp"). The pallet which is actually in use for a given part is specified by the attribute "px" of PART;
- PARTs can be made of (and can be components of) other PARTs: the aggregation class COMPRISES describes this situation;
- a machine can be a HANDLING-MACHINE or a MACHINE-TOOL.

Due to graphical reasons, the D-schema reports only the identifiers and the attributes which are relevant for the control description. The complete specification is given by Fig. 18.

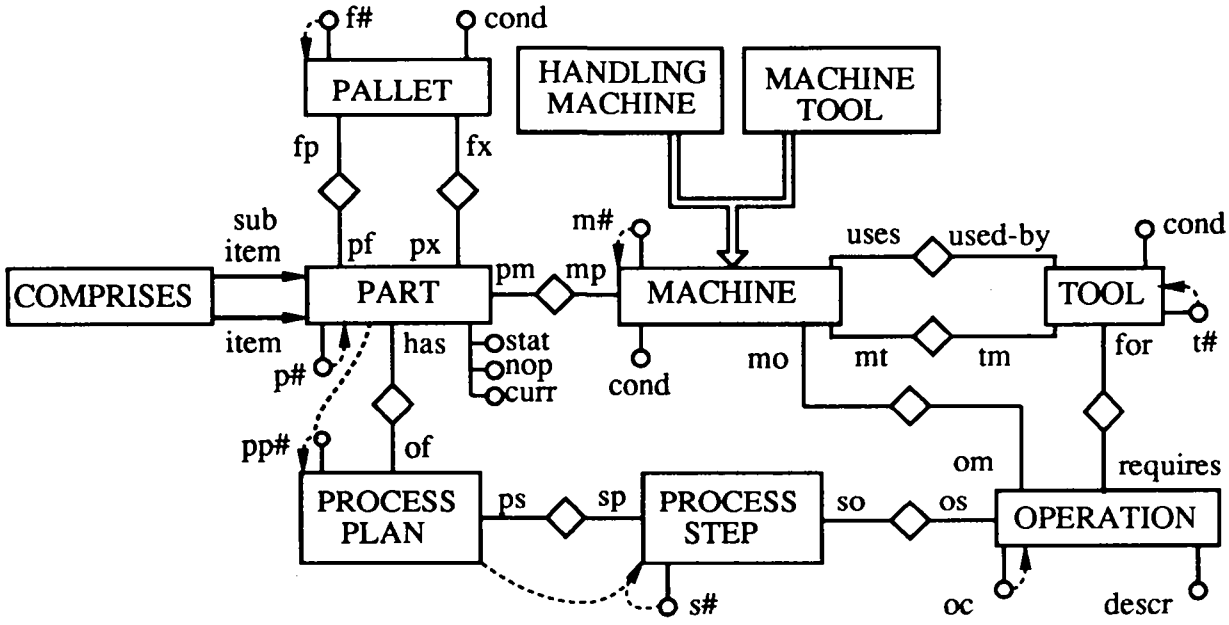


Fig. 17: The D-schema of the FMC Organisation Unit

class PALLET **identifier** f#
  member attributes f#: **integer**; cond: ('available', 'working'); fp(1,n): PART **inverse of** pf;
    fx(0,1): PART **inverse of** px
class PART **identifier** p#; part-id
  member attributes p#, curr, nop, lead-time: **integer**; stat: ('ready', 'mounted', 'working', 'worked', 'completed');
    p-name,part-id: **string**; has(0,1): PROCESS-PLAN **inverse of** of; pf(1,n): PALLET **inverse of** fp;
    px(0,1): PALLET **inverse of** fx; pm(0,1): MACHINE **inverse of** mp
class COMPRISES **composed of** (item: PART, sub-item: PART) **member attributes** level, qty: **integer**
class PROCESS-PLAN **identifier** (of,pp#)
  member attributes pp#: **integer**; ps(1,n): PROCESS-STEP **inverse of** sp; of (1,1): PART **inverse of** has
class PROCESS-STEP **identifier** (sp,s#)
  member attributes s#: **integer**; NC-cd, mch-cd: **string**; so: OPERATION **inverse of** os;
    sp: PROCESS-PLAN **inverse of** ps
class OPERATION **identifier** oc
  member attributes oc: **integer**; o-name: **string**; requires: TOOL **inverse of** for;
    om(1,n): MACHINE **inverse of** mo; os(1,n): PROCESS-STEP **inverse of** so
class MACHINE **identifier** m#
  member attributes o#: **integer**; cond: ('available', 'working'); mo(1,n): OPERATION **inverse of** om;
    uses(1,n): TOOL **inverse of** used-by; mp(0,1): PART **inverse of** pm; mt(0,1): TOOL **inverse of** tm
  subset HANDLING-MACHINE **of** MACHINE **member attributes**...
  subset MACHINE-TOOL **of** MACHINE **member attributes**...
class TOOL **identifier** t#
  member attributes t#: **integer**; cond: ('available', 'working'); used-by(1,n): MACHINE **inverse of** uses;
    tm(0,1): MACHINE **inverse of** mt; for(1,n): OPERATION **inverse of** requires

Fig. 18: The Specification of the FMC D-schema

At this point, the D-schema must be completed taking into account the life cycles of the objects referred to by the activities of the organisation unit. Life cycles describe the intrinsic behaviour of system components, i.e. the possible execution sequences of basic activities (activities which manage a single component and change the component state). Such activities are usually good candidates to become methods of the classes which describe the system components. The component state must be expressed by an attribute of the class and the component evolution must be represented by an object behaviour net which describes class methods. Pre- and post-states are in turn expressed by means of views on the D-schema.

Figure 19 shows the object behaviour nets for MACHINE, PALLET, PART, and TOOL objects which are relevant for the process Execute of Fig. 15.

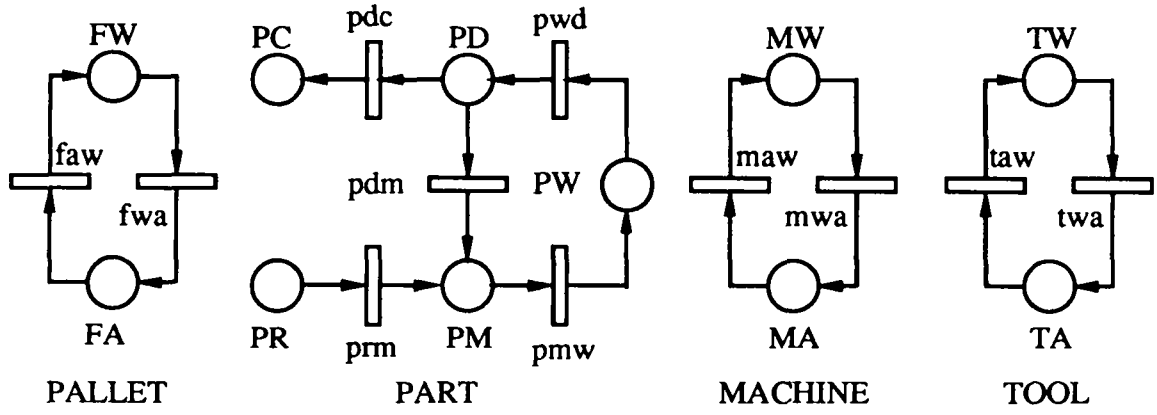


Fig. 19: PALLET, PART, MACHINE, and TOOL Object Behaviour Nets

```
class PALLET ...                                    methods faw(p|PART); fwa
class PALLET implementation
    view FA(f|PALLET) where f.cond='available'; view FW(f|PALLET) where f.cond='working';
    method faw(p|PART) accepts FA(f) return FW(f) exec f.Update (cond:'working', fx:p) end
    method fwa accepts FW(f) return FA(f) exec f.Update (cond:'available', fx:nil) end

class PART ...                                      methods prm(f|PALLET); pmw(m|MACHINE); pwd; pdm; pdc
class PART implementation
    view PR(p|PART) where p.stat='ready';           view PM(p|PART) where p.stat='mounted';
    view PW(p|PART) where p.stat='working';         view PD(p|PART) where p.stat='worked';
    view PC(p|PART) where p.stat='completed';
    method prm(f|PALLET) accepts PR(p) return PM(p) exec p.Update (stat:'mounted', px:f, curr:1) end
    method pmw(m|MACHINE) accepts PM(p) return PW(p) exec p.Update (stat:'working', pm:m) end
    method pwd accepts PW(p) return PD(p) exec p.Update (stat:'worked', pm:nil, curr:curr+1) end
    method pdm accepts PD(p) return PM(p) exec p.Update (stat:'mounted') end
    method pdc accepts PD(p) return PC(p) exec p.Update (stat:'completed', px:nil) end

class MACHINE ...                                   methods maw(p|PART;t|TOOL); mwa
class MACHINE implementation
    view MA(m|MACHINE) where m.cond='available'; view MW(m|MACHINE) where m.cond='working';
    method maw(p|PART;t|TOOL) accepts MA(m) return MW(m) exec m.Update(cond:'working',mp:p,mt:t) end
    method mwa accepts MW(m) return MA(m) exec m.Update (cond:'available', mp:nil, uses:nil) end

class TOOL ...                                      methods taw(m|MACHINE); twa
class TOOL implementation
    view TA(t|TOOL) where t.cond='available';       view TW(t|TOOL) where t.cond='working';
    method taw(m|MACHINE) accepts TA(t) return TW(t) exec t.Update (cond:'working', tm:m) end
    method twa accepts TW(t) return TA(t) exec t.Update (cond:'available', used-by:nil) end
```

Fig. 19: PALLET, PART, MACHINE, and TOOL Object Behaviour Nets (continued)

The next step of this task is the construction, for every process of the object organisation unit, of the process net which describes the behaviour of that process in compliance with the conceptual model. All the process and behaviour nets constitute the F-schema of a given organisation unit.
To construct a process net, the activities which are components of the process must be modelled as conceptual routines. Input and output requisites of an activity must be expressed by means of views on the D-schema, taking into account the views which describe the state of the objects that are manipulated by the activity. Finally, the operations described by the activity specification must be expressed by calling suitable methods on that objects.

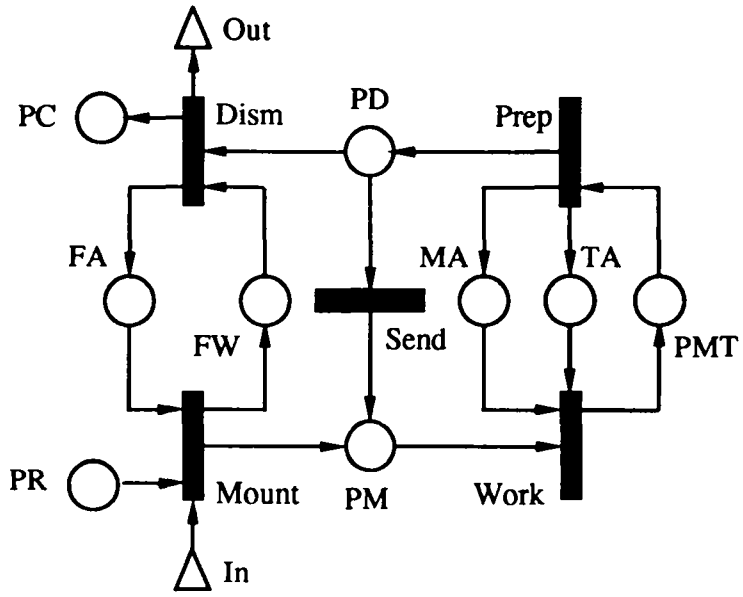
The composition of the routines can be carried out taking into account event sentences of the process (to discover causal relationships between the routines) and their input and output places. Corresponding places (i.e. messages expressing the same system state and views describing the same objects in the same state) must be recognised. The process net is then obtained by superimposing corresponding input and output places of related routines. As in the Organisation Analysis phase, the methodology suggests an outside-in strategy which starts from the process interface (consisting of "external" places, i.e. messages and views exchanged with the external world or other processes), inserts the routines in the schema which are enabled by the external places, then the routines enabled by them, and so on.
For instance, starting from the description of the Execute process reported in Fig. 15, the following routines can be recognised:

- Mount: A part (with a given part number 'np' specified in the message In and ready to be worked) is mounted on a pallet (which must be 'available');
- Work: A part is processed via the execution of a sequence of specific process steps, each referring to one operation to be performed by a machine with a proper tool. The actual operation to be executed is specified by the number stored in the attribute "curr" of PART;
- Prep: At the end of each operation, the part must be prepared for the next operation; the routine returns the machine and the tool used in the previous operation, and selects the next process step by incrementing the attribute "curr";
- if all the operations on the part have been completed (the current value of "curr" is greater than the total number of operations to be executed, stored in the attribute "nop" of PART) the part is dismounted and is stored into an output buffer (routine Dism); otherwise, the routine Send sends the part back into the cell for further processing.

In Fig. 20 the process net which models the Execute process (i.e. the activities of the Flexible Manufacturing Cell) is illustrated. Buffers containing parts at different stages of manufacturing (parts 'ready' to be worked, 'mounted' on pallets, 'worked', and 'completed') are represented by means of the views PR, PM, PD, and PC, respectively, views FA, FW and MA represent 'available' and 'working' pallets and 'available' machines, respectively (all these views have been defined in Fig. 19). The view PMT(p,m,t) specifies a part p which is actually worked on a machine m with a tool t. The message Out gives the number of a part whose manufacturing process has been completed.
The last step of this task is the validation of the process nets. Its aim is to validate the developing project before the implementation begins so that modifications can be made to solve inconsistencies and ambiguities. Since in the M*-OBJECT approach this step uses a simulation tool, this tool is described in Section 9 of the paper.

7. The Environment Design and Integration Tasks

The Environment Design task integrates the descriptions of all the organisation units of the environments to be analysed. As usual, both static descriptions (local D-schemata) and dynamic descriptions (local F-schemata) are integrated to produce the conceptual representation (environment schema) at the environment level. A further integration step leads to the conceptual schema.
The integration of local D-schemata and of environment D-schemata requires more care than the intra-unit aggregation carried out in the Local Design task because users belonging to different units and environments may have different perceptions of the same concepts, giving rise to different representations. Moreover, local and environment D-schemata can be independently developed by different designers at different times. As a consequence, D-schemata integration at the organisation unit and environment levels are critical tasks [6].
After the conceptual D-schema has been produced, local and environment D-schemata must be restructured to make them coherent with the representation of data chosen in the conceptual D-schema.
The co-ordination of process nets analyses the communications (both data and messages) between organisation units and environments, and between them and the external world. Since, at this level, the internal structure of processes (as nets of places and routines) is irrelevant, during the analysis each process is represented as a macro-process with the external places of the process as its input and output places. For instance, Fig. 22 illustrates the macro-process corresponding to the Execute process for the FMC unit.



message In(np: **integer**) **message** Out(np: **integer**)

view PR... ; **view** PM... ; **view** PD... ; **view** PC...; **view** FA...; **view** FW...; **view** MA...;
view PMT(p|PART, m|MACHINE, t|TOOL)
where p.stat='working' **and** m.cond='working' **and** t.cond='working' **and** p<pm>m<mt>t

routine Mount [**accept** In(num), PR(p), FA(f) **return** PM(p), FW(f)]
guard p.p#=num **and** p<pf>f **exec** p.prm (f); f.faw (p) **end** Mount

routine Work [**accept** PM(p), MA(m), TA(t) **return** PMT(p,m,t)]
use x|PROCESS-PLAN, y|PROCESS-STEP, z|OPERATION
guard p<has>x<ps>y<so>z<om>m **and** m<requires>t **and** p.curr=y.s#
exec p.pmw (m); m.maw (p,t); t.taw (m) **end** Work

routine Prep [**accept** PMT(p,m,t) **return** PD(p), MA(m), TA(t)] **exec** m.mwa; p.pwd; t.twa **end** Prep

routine Send [**accept** PD(p) **return** PM(p)] **guard** p.curr≤p.nop **exec** p.pdm **end** Send

routine Dism [**accept** PD(p), FW(f) **return** PC(p), FA(f), Out(num)] **guard** p.curr>p.nop **and** p<px>f
exec num:=p.p#; p.pdc; f.fwa **end** Dism

Fig. 20: The Process Net for the Flexible Manufacturing Cell

The overall architecture of the Local Design task is given by Fig. 21.

foreach organisation unit:

Build the Local D-schema (Data schema)

1. aggregate form and record format D-schemata to obtain a draft D-schema
2. foreach activity: aggregate the activity D-schema to the draft D-schema to obtain the new draft D-schema
3. construct the object behaviour nets of relevant objects
4. complete the local D-schema with methods specification

Design the Local F-schema (Functional schema)

5. foreach process of the object organisation unit:
   - 5.1 model each activity of the process and identify the external places
   - 5.2 construct the process net
   - 5.3 validate the process net

Fig. 21: Local Design Steps

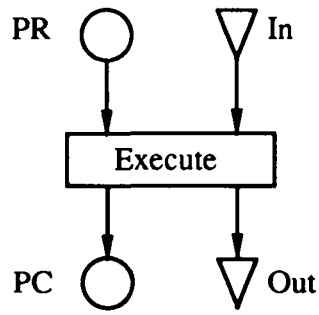


Fig. 22: The Execute Macro-process

By composing the macro-processes of an environment the organisation functions of that environment will be expressed as functional nets, i.e. nets of places and macro-processes. These nets constitutes the F-schema of the object environment. A further composition step, starting from the nets of all environments to be analysed, generates the conceptual F-schema of the object enterprise.
The composition of macro-processes can be carried out as described in the Local Design task for routines composition. It must be noted that functional nets are no more Petri nets because they do not specify in which way (for instance, which order) messages and views are manipulated. Therefore, these nets cannot be executed and can be considered as a data flow description of the environment functions. Moreover, they can be related to the original organisation nets constructed in the Organisation Analysis phase, the basic difference consists in the complete specification of the "interfaces" between the environment functions in terms of messages and data defined on the conceptual D-schema.
The overall architecture of the Environment Design and Integration tasks is given by Fig. 23.

Build the Environment D-schemata

foreach environment to be automated:

1. choose a local D-schema as the draft D-schema
2. repeat:
   2.1 integrate a local D-schema into the draft D-schema to obtain the new draft D-schema
   2.2 restructure the draft D-schema
   until all local D-schemata have been processed
3. restructure the environment D-schema

Build the Conceptual D-schema

4. integrate the environment D-schemata into the conceptual D-schema
5. restructure the conceptual D-schema

Build Environment and Conceptual F-schemata

foreach environment to be automated:

6. foreach process of the object environment, introduce a macro-process
7. construct the functional nets of the object environment
8. construct the functional nets of the enterprise

Fig. 23: View and Environment Integration Steps

8. The Implementation Design Phase

The Implementation Design phase translates the conceptual description of the object enterprise (the conceptual schema) into the specific data model supported by the DBMS in use. It is composed of two major tasks: Logical Design and Physical Design.

The aim of the Logical Design task is to translate the conceptual schema into a schema in which data structures are expressed according to the logical data model of the DBMS chosen for the implementation.
The aim of the Physical Design task is to provide database designers with a framework of design decisions that would lead to a "reasonable", i.e. more or less optimised, physical design. Criteria are given for choosing an "efficient" logical and physical representation among several possible alternatives that can be derived from the same conceptual schema. No metrics exist to quantify these design properties which are prone to subjectivity.
The conceptual schema, as the input to the Implementation Design phase, is an executable specification of static, dynamic, and behavioural enterprise features. More specifically, the conceptual schema comprises:

- the conceptual D-schema (data schema), which describes all the data relevant for the object enterprise;
- a set of environment D-schemata, in which are represented the data of the different enterprise environments of the CIM system. Usually, an environment D-schema is a consistent and commonly accepted view of data for a significant part of the enterprise;
- a set of process and functional nets for each environment.

D-schemata refer to the structure of data and to the integrity constraints data must satisfy and will be translated in terms of the target data model.
In the ANSI/X3/SPARC proposal of an architecture for DBMS [16], a DBMS is organised according to three levels of data description, external, conceptual, and internal. At the external level, a database is perceived as a collection of user views, each defined by an external schema. The conceptual level provides a high-level representation of the whole database. To avoid any ambiguities, in the following this level will be referred to as the logical level, and the data representation will be called the logical schema of the database. At the internal level, a machine-dependent description of the database, defined by an internal (or physical) schema, is provided. In this paper (and in several practical cases) both at the external and logical level the same model of data, e.g. the relational model, and the same data definition and manipulation language, e.g. the SQL database language [13], are used.
Typically, in the Logical Design task the conceptual D-schema is translated into the logical schema of the whole database and the environment D-schemata are translated into external schemata.
In the actual version of the M*-OBJECT methodology, the relational model [13] has been adopted as the target model. External and logical schemata are then expressed as relational schemata, and suggestions to convert process specifications into SQL "skeleton" programs on the logical schemata (obtained in the Logical Design task) are introduced. Skeleton programs deal with the manipulation of the relational database only, then they have to be integrated with other procedures dealing, for instance, with human-computer integration, security, and so on. The Physical Design task constructs a physical schema, which specifies allocation parameters and access paths, i.e. a set of clustered and secondary indexes on the physical relations.
Since the translation from an object-oriented model to the relational model is now a well-understood problem [see, e.g. [17]), this phase will not be discussed in this paper. Details on the implementation design are given in the original report of the M* methodology [18].

9. The Design Tool

This section briefly describes an experimental graphic-oriented tool for specification and validation of information system design, called M*Designer, which supports the Conceptual Design phase.
M*Designer is a graphical tool based on the WIMP paradigm (Windows, Icons, Menus, Pointing devices) to govern user interaction with larger systems on workstations. The user shows "what to do" by manipulating visual representation of objects. This approach has several advantages: users can permanently see the data they are working on, they can check results of their actions, and they can easily perform manipulations to modify the graphical representation by activating functions using menus, labelled buttons, and dialog boxes.
The M*Designer environment consists of the following components (which are integrated by means of a central repository):

- SCHEMA EDITOR - It is a visual data specification interface which allows the designer to build a conceptual schema from scratch. The editor provides two modes of operation which correspond to different windows.

  The PICTURE MODE allows to build schemata by picking up graphical icons from a palette (which contains symbols corresponding to object-oriented data constructs) and putting them into the workspace provided by the picture window. Since the editor is syntax directed, incorrect schemata (from a syntactic point of view) cannot be produced (for instance, it is not possible to connect two relationship symbols by an arc).

  The TEXT MODE allows to specify schema objects by entering textual definitions through object templates (different object templates correspond to different object-oriented constructs). Obviously, the two modes are equivalent and the editor keeps them synchronised.

  The editor provides a simple backup facility with UNDO and REDO commands; UNDO acts on the latest operations, while REDO reinstates operations reversed with UNDO. LOAD/SAVE commands allows to backup/restore a complete schema.
- SCHEMA EVOLUTION - It is a module which is invoked during schema design to change the class definitions and/or the structure of hierarchies. Types of changes which have been implemented include creation, deletion, renaming of classes, attribute and relationships, and alteration of hierarchies. Graphic icons of classes and relationships can be repositioned in the picture window to improve visual readability.
- NET EDITOR - It is a visual editor for the specification of process nets which is equivalent to the schema editor (it provides the designer with the same functionalities). The two editors are synchronised in the sense that objects which appear in the textual description of views and routines of a net must be defined in the associated data schema.
- SIMULATOR - This component is an executing environment (based on production rules) in which the system database is stored into a working memory as a set of facts. A conceptual interpreter translates process routines into rules of the form:

  if «condition» **then** «action»

  in which the «condition» part consists of the conjunction of the predicates which appear in the input view specifications and the guard clause of the routine, the «action» part contains the translation of routine actions into update operators on facts.

  Rules are then scanned by an inference engine which selects all the rules that are satisfied (enabled) by the set of facts; this step is based on the RETE algorithm for pattern matching [19], modified to take full advantage of the particular form of process nets (in which the set of rules that can be activated after a rule has been applied is given by the net structure). One of the enabled rules is (randomly) selected and applied, and its consequences (described in the «action» section) update the existing set of facts.
- ANIMATOR - The simplest way to validate the system specification is to follow its behaviour by watching the system state as time evolves. The animator is a module which supports all the interactions with users and monitors the execution of simulation steps by calling the simulator. Users can then follow in an interactive way routine's "firing" and receive all the data concerning the evolution of both the database state and the event structure.
- INITIALISER - It is a visual module which is activated by the animator when the validation session begins. Object frames allows users to input into the system database the set of objects and active messages which will be used as initial conditions in the system simulation. All input data are checked against class specifications to verify that they satisfy all the integrity constraints expressed in the schema.

The system has been implemented in C++ using a UNIX environment. It runs on SUN workstations.

10. Conclusions

M*-OBJECT is a step-by-step methodology applied to manufacturing information systems analysis and design. It has mainly been developed for the needs of CIM consultants and industrial engineers. It has been tested on several real-world case studies. From these case studies, some advantages and disadvantages of M*-OBJECT have been noticed from a practical point of view.

The advantages of M*-OBJECT include:

- it is a pragmatic, step by step methodology which is complete, i.e. it covers all database design phases from general requirements specification and analysis to detailed physical database schema specification;
- it is based on good modelling tools (both at the organisation and conceptual levels);
- it deals with static, dynamic and behavioural properties of data, providing tools to validate the conceptual data schema obtained;
- it involves an efficient and detailed system analysis and modelling phase useful to detect anomalies in the operations of the functions of the enterprise and to prepare the automation phase;
- it really helps communication between administrative and engineering personnel of the company in terms of systems modelling and information modelling and processing.

Some disadvantages of M*-OBJECT are:

- it is a complex methodology involving different modelling techniques;
- it is prone to human subjectivity in several steps of both the organisation analysis and the conceptual design phase;
- it must be interfaced with integrated tools dealing with complete CIM system design;
- more general simulation tools are needed. In fact, since the time concept has not yet been introduced, the PDN model admits only a "qualitative" simulation, which shows the evolution of database states and process net markings.

Obviously, the complexity of the method is related to one of its qualities: completeness. Information modelling and database design in production environments is not a trivial task, therefore models and tools to support this activity must have some degree of complexity. From the drawbacks mentioned, it is obvious that M*-OBJECT must be used by experienced people. This remark undertakes the implementation of an expert advisor system to be used as a tutorial advisor for teaching the principles of the methodology and as an expert analyst for demonstrating the treatment of some experimental case studies.

Concerning the introduction of the time concept in the PDN model, the extended model which is under investigation will allow a description of the dynamic behaviour that takes into account not only the state evolution but also the duration of the actions performed by the system. In this way, several quantitative time-dependent parameters, such as throughput, can be modelled [20].

11. References

[1] Vail P.S. *Computer-Integrated Manufacturing*, PWS-Kent Publishing Company, 1988.

[2] DiLeva A., Vernadat F., Bizier D. "Information System Analysis and Conceptual Database Design in Production Environments with M*" *Computers in Industry* **9**: 183-217, 1987.

[3] DiLeva A., Vernadat F., Bizier D. "Enterprise Analysis and Database Design with M*: A Case Study" *Computers in Industry* **11**: 31-52, 1988.

[4] Rumbaugh J. et al. *Object-Oriented Modelling and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[5] DiLeva A., Giolito P., Vernadat F. "The M*-OBJECT Methodology for Information System Design in CIM Environments: The Organisation Analysis Phase" Res. Rep. INRIA and Dipartimento di Informatica, Università di Torino, 1993.

[6] Batini C., Ceri S., Navathe S.B. *Conceptual Database Design: An Entity-Relationship Approach*, Benjamin Cummings, 1992.

[7] Albano A. et al. "An Overview of Sidereus, a Graphical Database Schema Editor for Galileo" In Schmidt J.W. et al. (eds.) *Advances in Database Technology - EDBT '88*, Springer-Verlag, Berlin, 1988.

[8] Kuntz M., Melchert R. "Pasta-3's Graphical Query Language: Direct Manipulation, Cooperative Queries, Full Expressive Power" *Proc. of 15th Int'l Conf. on Very Large Data Bases*, Amsterdam, 1989.

[9] Larson J. "A Visual Approach to Browsing in a Database Environment" *IEEE Computer* **19**, 1986.

[10] Reiner D. et al. "A Database Designer's Workbench" In Spaccapietra S. (ed.)*Entity-Relationship Approach*, North-Holland, Amsterdam, 1987.

[11] Auddino A. et al. "SUPER: A Comprehensive Approach to DBMS Visual User Interfaces" *IFIP 2nd Conf. on Visual Database Systems*, Budapest, Hungary, 1991.

[12] Reisig W. "Petri Nets in Software Engineering" In Brauer W. et al. (Eds.) *Petri Nets*, LNCS 255, 63-96, Springer-Verlag, Berlin, 1987.

[13] Date C.J. *An Introduction to Database Systems*, Vol. I, 4th ed., Addison-Wesley, Reading, MA, 1986.

[14] Batini C., Demo B., DiLeva A. "A Methodology for Conceptual Design of Office Data Bases" *Information Systems* **9**(3/4): 251-263, 1984.

[15] DeAntonellis V., Demo B. "Requirements Collection and Analysis" In Ceri S. (Ed.) *Methodology and Tools for Database Design*, North-Holland, Amsterdam, 1983.

[16] ANSI/X3/SPARC Study Group on Data Base Management Systems "Interim Report" *ACM SIGMOD Bulletin* **7**(2), 1975.

[17] Markowitz V.M., Shoshani A. "Representing Extended Entity-Relationship Structures in Relational Databases: A Modular Approach" *ACM TODS* **17**(3), 1992.

[18] DiLeva A., Vernadat F., Bizier D. "M*: A Methodology for Information System Analysis and Database Design of Production Systems" Res. Rep. NRCC 26204, ERB-1000, National Research Council of Canada, Ottawa, K1A 0R8, Canada, 1987 (98 pp.).

[19] Forgy C.L. "A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem" *Artificial Intelligence* **19**(1), 1982.

[20] Balbo G. et al. (Eds.) *Proc. Int. Workshop on Timed Petri Nets*, Torino, Italy, 1985.

Unité de Recherche INRIA Lorraine
Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Antenne de Metz
Technopôle de Metz 2000 - Cescom - 4, rue Marconi - 57070 METZ (France)

Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399