Beyond the Object Oriented Analysis

Luca Dan Serbanati
Freelance consultant in informatics

Via Piero Foscari 40, Pa. A, Sc. C, In. 10
00139 Roma
ITALY

Abstract: Developing a computer-based system is essentially a transformation process of a conceptual representation emerging from the user's informal requirements into an operational system. As known, the user's requirements in the early phases of the system development are imprecise, incomplete, often inconsistent or partial wrong, themselves in evolution. The impossibility of having from the start an accurate description of the user's application universe is mostly due to the analyst's particular universe of discourse in which the user recognizes with difficulty how his or her real and future systems were mapped. It is visible for the last two decades a permanent tendency of approaching the analyst universe to the user's real world. It is very true that the object-oriented (OO) approach brought nearer the two points of view, but the OO paradigm is still too general or abstract to be easily operational for non-professionals. Moreover, the OO paradigm is inadequate for many business applications or at least for some components of them. That is why, to improve the impact of the OO paradigm and to tune its undoubted breakthroughs in analysis, more nuanced OO analysis methods are needed.
The paper introduces a method of system analysis for eliciting the requirements and functionalities of systems in the early stages of system development life cycle. The method is aimed to bring the analyst's universe nearer to the user. It differs from them in some premises and in their argumentation as well as in requirements specification languages.
Keywords: System development life-cycle, analysis method, structured analysis, object oriented analysis, conceptual approach, conceptual decoupling of concerns, object facet, facet cluster, algebra environment, DFD, ERD, STD.

Luca Dan Serbanati received the M.Sc. degree in Process Control from the Polytechnical Institute of Bucharest in 1966, the M.Sc. degree im Mathematics from Bucharest University in 1972, and the Ph.D. degree in Computer Science from the Polytechnical Institute of Bucharest in 1980. He has taught for 20 years at the Polytechnical Institute of Bucharest courses in theoretical informatics, programming languages, compiler design, and software engineering.
Since 1990 he has been working in Italy as independent consultant in informatics providing software products for industry, consulting and training in computer programming, system analysis and design, software house organization, and software technology migration to industry. He is author of the monography Programming Languages and Compilers (Romanian Academy Press ,1987) and of several textbooks and lecture courses in informatics and computer programming. His latest book titled Integrating Tools for Software Development is on the software process modelling and software engineering environments and was published in 1992 by the prestigious Yourdon Press Computing Series. He is currently interested in formal models and curricula in software engineering,conceptual-oriented system analysis, object-oriented software technologies, and CASE tools. He is member of IEEE,ACM , and IEEE Computer Society.

I. Introduction

Developing a computer-based system is essentially a transformation process of a conceptual representation emerging from the user's informal requirements into an operational system. During this process, transitions from the user's real world to that of the computer and from there backwards in the real world are needed. But the two universes are so far apart from each other that such straight transitions would be abrupt and unexpected in their consequences. That is why all system development life cycle models suggest intermediate conceptual universes to be smoothly crossed. According to various methodologies the development process and the project itself should cross, one after another, such universes as: informal requirements, functions, data structures, algorithms, and programming languages. Throughout these transitions, the developer should bear in mind and continuously refine the user's requirements specification which describes as accurately as possible the *application universe:* that imaginary (often visionary) world born in the user's mind and aimed to be the solution for the user's present problems. All subsequent conceptual universes should maintain straight and permanent links with the user requirements specification, in fact mappings between the deliveries of the intermediate stages of the development process and the user's requirements.

During the last decade, to the two previous main objectives of any methodology: how to manage the universe transitions and how to conserve the user's requirements semantics throughout the development process, another objective, obvious for any engineering technology, was added: how to reuse already existent components in the new system.

As known, the user's requirements in the early phases of the system development are imprecise, incomplete, often inconsistent or partial wrong themselves in evolution. The impossibility of having from the start an accurate description of the user's application universe is mostly due to the analyst's particular universe of discourse in which the user recognizes with difficulty how his or her real and future systems were mapped. For the user, the application universe consists of images, sensations and much acquired practical knowledge about real things, beings, events, and their relationships. The analyst works with collections of data flows, data files, system states, entities, and attributes. Where the user is accustomed to view automated and manual operations or certain activities able to accomplish a certain behavior of the system, the analyst finds functions, state transitions, processes, services, or methods.

It is visible for the last two decades a permanent tendency of approaching the analyst universe to the user's real world. It is very true that the object-oriented (OO) approach brought nearer the two points of view, but in our opinion there are other steps to be taken. Many users find the OO approach too flat, still far away of their application universe. The OO universe uniformity, which is so important for the analyst, is precisely what bothers the user. In other words, the OO paradigm is still too general or abstract to be easily operational for non-professionals. Moreover, the OO paradigm is inadequate for many business applications or at least for some components of them. While it matches perfectly the user interface prerequisites, for other components as database management and process control it is less suggestive and less efficient than, for instance, the transactional approach. That is why, to enhance the impact of the OO paradigm and to tune its undoubted breakthroughs in analysis more nuanced OO analysis methods are needed. These methods should also incorporate many aspects of the conventional analysis methodologies. Valuable approaches (see [1], [2]) could be found in many recent research works.

The paper introduces some ideas aimed to bring the analyst's universe closer to the user. The approach is based on our research work on the software process modeling done in the last '80s [3]. This work has shown us how poor is our analysis workshop when the application universe is complex, as the software development process is. The method we have used for software process modeling is similar to other methods and methodologies emerged last years. It differs from them in some premises and in their argumentation as well as in specification languages aimed to be used by analysts together with well-known analysis tools like data flow diagram (DFD), entity-relationship diagram (ERD), data dictionary (DD), Petri nets (PN) and state transition diagram (STD).

In Section 2 the method paradigm is introduced. Section 3 discusses the method steps and illustrates by means of some examples how the method addresses the specification language.

2. The Basic Model

We consider a computer-based system as several agents (people, organizations, computers, physical or abstract data processing devices) performing a variety of activities (sorting, monitoring, I/O, controlling, etc.) that act upon the system objects (raw- materials, intermediate forms of the final product, orders, alarm events, peripheral relays, product documentation,etc.) eventually using some specialized tools so as to accomplish the system objectives (Figure 1). In this framework four entity types evolve:

- *object* = a passive entity used and/or modified by other entities;
- *activity* = an entity able to change the system state usually by object transformations;
- *agent* = a system entity able to carry out one or more activities according to its skills;
- *tool* = a specialized activity which is carried out by specialized agents and aims at satisfying object processing needs of several activities.

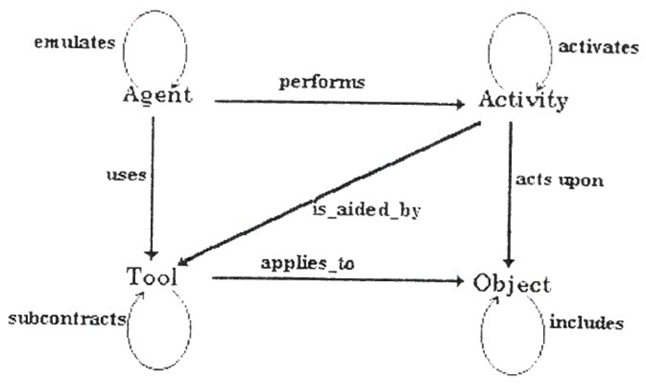


Figure 1. Conceptual Schema of the System Entity Types

The paradigm was suggested by engineering environments where entities involved have various roles: raw-materials and various product intermediate forms, prescribed technological activities, workmen, and necessary tools. One could note that some entities change their role when the environment changes: a workman becomes an object when acquires new skills, an activity becomes a simple tool in an automated environment, an inactive processor becomes an agent when it receives a program for execution, etc. Moreover, when the entities are complex, by refining steps, that is knowing more about their structure or functioning, their role in our mind modifies. It is possible that such an entity becomes a system in itself. A complex instrument becomes a collection of parts and tasks carried out by each part, a factory could be viewed either as some people doing specialized activities or as an activity that processes some raw- material, and so on. The two views are complementary and both are valid. That is why we have to precisely delineate both the environment in which we are interested and what we want to consider or know about the system under analysis.

As known, during analysis process, more conceptual levels of various refinement degrees should be outlined. All entities the analyst identifies for bringing together must belong to the same conceptual level. An entity belonging to a conceptual level should:

• preserve both its responsibilities and its interfaces with other entities throughout the model functioning;

• have its internal structure and functionality unknown or irrelevant to the momentary concern that has delineated the conceptual level.

These two conditions are important for stabilizing the characteristics of the components belonging to any conceptual level of approaching the system. When they hold, it is much easier for the analyst to classify system components according with the user's preferences and intuition in one of the four entity types.

Although agents and tools are important entity types in our method, in the following we concentrate on objects and activities.

2.1. System Objects

2.1.1 Views on System Objects

The view we propose on objects is in many aspects (encapsulation, inheritance) similar to the OO view. The object concept is a means for more accurately modeling the application universe, but only certain entities that populate this universe are objects according to our approach. Although tools, agents and some aspects of activities could be objects according to the OO approach, we claim that it is important to differentiate among the roles of various components in the application universe and then, in the user requirements specification. This way, in the system formal specification we may introduce concepts belonging to the user's application universe keeping the analyst specification near to the user's view of the system. In our approach the analyst also benefits from more flexibility in specifying and manipulating objects. Figure 2 hints at our approach to objects. According to the analyst momentary interest there are two views on objects: the user's view and the designer's view.

The *user's view* introduces an object entity to other entities at the same conceptual level. This view has two sections:

- *operations:* functions evoked from the object environment and aimed to modify or only to inform about the object state;

- *links*: relationships with other objects situated at the same conceptual level.

The *designer's view* opens the door to the internal structure and functioning specification, and thus, to refinement operations. This view has three sections:

- *attributes*: simple properties of the object;
- *facets*: clustered models of the object;
- *constraints*: predicates on attributes and facets.

A simple object could be completely specified using some attributes. More difficult is to describe complex objects such as large structures or environments, highly structured files, or modern machines. To master these objects and their evolution one must resort to a concept that we found especially useful in modeling: facet clustering.

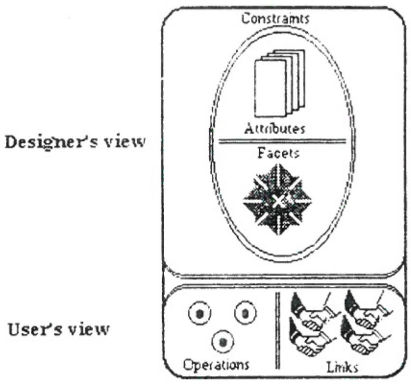


Figure 2. A Hint of the Object Concept

2.1.2. Facet Cluster

It is known that human knowledge is model-oriented. Our mental representation of a real, known object is strongly shaped by some metamodels or paradigms we know and which, at that instant and for some reason, we associate with the object. As a rule, a metamodel is a systemic view of a class of objects, a generic frame that represents the object (and many others similar to it) as a structure made up of interrelated components, themselves objects to be modeled. Initially, when we learned or saw for the first time the object, or later, when we had to update our knowledge of the object, these metamodels were confronted with some essential properties of the object and, if they matched these properties, they were instantiated in such a way as to precisely become models of the object.

Of course, in the modeling process we neglect some aspects of the object, retaining the most relevant ones from a certain point of view or at a certain abstraction level. If the object is sufficiently complex, one may create "multiple" models of it. Once relevant aspects are pinpointed in several models, the models may be studied and modified independently and more easily than the original object. These models can be compared with the facets of a spatial object: from a point in space we see a facet of the object, from another point, another, different facet, and so on. When we analyse a real object, we study it only through the facets we know. When we synthesize or design a new object the only tools we can use for verifying if it matches some essential properties are the facets brought together to build it. Among facets there should be all models of the object that may be useful to other system entities in order to study, modify, or dialogue with the object. If the facet set is complete, that is all useful views on the object were identified by the analyst, the set represents the object itself. Even if the object is more complex and other views on it are possible, they are not useful for analysis. We shall use the term *facet* to designate such a model of a complex object.

Since the models underlying facets are to a certain extent independent, each facet becomes a separate object (a *subobject* of the original object) that can be studied. If the model is complex, it is multifaceted too; its facets may also be multifaceted objects; and so on. Thus, model-based multifaceting of objects leads us to a hierarchical decomposition of them. The subordination of subjects to their *superobjects* represents a refinement relationship and is essential for describing objects of high complexity. Of course, such a hierarchy must have terminal objects: the subjects without a relevant internal structure (only their attributes are sufficient for specifying them).

For identifying an object's facets we have already presented a possible criterion: how the object is viewed, used, or modified by other entities in the system. Guidelines to analysts could be defined the criterion of identifying three levels almost ever relevant to approaching each facet. The first level, the deepmost one, called *semantic* grasps the meaning of the object expressed in various ways as: the activity the facet evokes (for instance, when the object is an activity plan or schedule, or when it represents a process description), a mathematical expression, or a simple comment describing the meaning the analyst intends for the object. The second level, which we called *structural,* captures the essential structure of the facet. It enters the object components in some standard, high-level structuring relationships as: sets, arrays, records, lists, trees, graphs, etc. The third level, we called it *textual*, captures the external presentations of the object, that is those graphical images or alphanumeric texts based on which the user will identify the object on the display screen or in a document. The textual level usually refers the structural facet for some essential information, for mapping it in graphical symbols, windows, icons, various drawings, or 2-D or 3-D graphical relationships. This additional information makes the object presentation more readable, more manageable, or more suggestive. Although a detailed description of the external presentation of the object is a later task of designing phase, the main traits of the user interface of the system and the user's constraints on this interface should be early expressed in the analysis delivery. In the level hierarchy we consider the structural approach as central: it contains only essential information liable to be formalized. The other two levels usually refer it.

As already stated, it is possible that one or several users may have different views of the same object, due to different occasional interests, structuring criteria, or levels of concern. It is worthy for an analyst to capture all facets that are relevant to the system functioning from the user's point of view. Because all these views refer the same object, it is obvious that the object facets should be clustered together by some relationships and consistency rules or procedures such that when a facet is modified, all the others will be automatically updated. We consider the *facet cluster as* the most suitable schema for modeling system objects at the analysis time. Each facet in the cluster is a model of the object that the cluster represents, an insight into that object.

2.1.3 Object type hierarchy

Beside encapsulation, in order to re-use the system entities, our method supports classes and inheritance. In our approach, an object is an *instance* of an object *type* and has all properties of that type. The instances of a type are distinct, that is they have proper identity, beside common properties. A type is a description of (both behavioural and structural) properties, a set of objects evinces. The collection of all instances of a type, available after a while, is the *class* of that type. The content of such a class may vary during the system functioning. A class may contain several subclasses of objects, specializations of the class. The objects in a subclass have some identical properties with those of the objects in the other subclasses. These properties are *inherited* from the class properties. But the objects in a subclass may also have different properties corresponding to the subclasses they belong to. However even if **is_a** is a fundamental relationship of object orientation, it is less useful for analysis than for design and implementation phases. At analysis time it is too early to have a complete **is_a** hierarchy of objects.

2.1.4 Object Interface (User's view)

The *object interface* is more relevant at different times than the object structure. It includes information on how the object acts as a whole. In our approach the object interface includes object operations and links.

The *operations* are object state transformers. An operation is characterized by its name (the operator), its arity, that is its number of operands, operand types and the result type (if any). The operations could change the attributes values or facets, or verify that some conditions hold in the object state. The operations are also means for introducing into an object transformational capabilities that are typical to activities. For instance, a finite-state machine approach could be used in object specification: the object with its attributes and facets is the state repository and the operations are the state transformers.

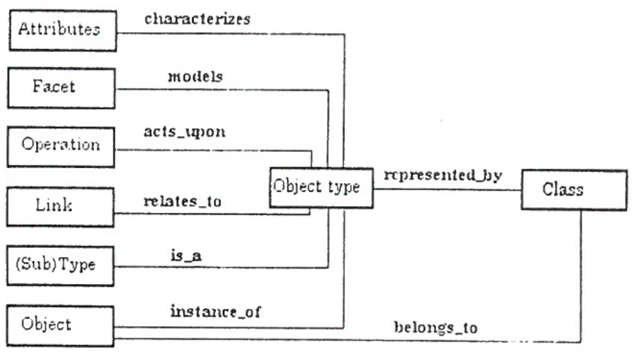
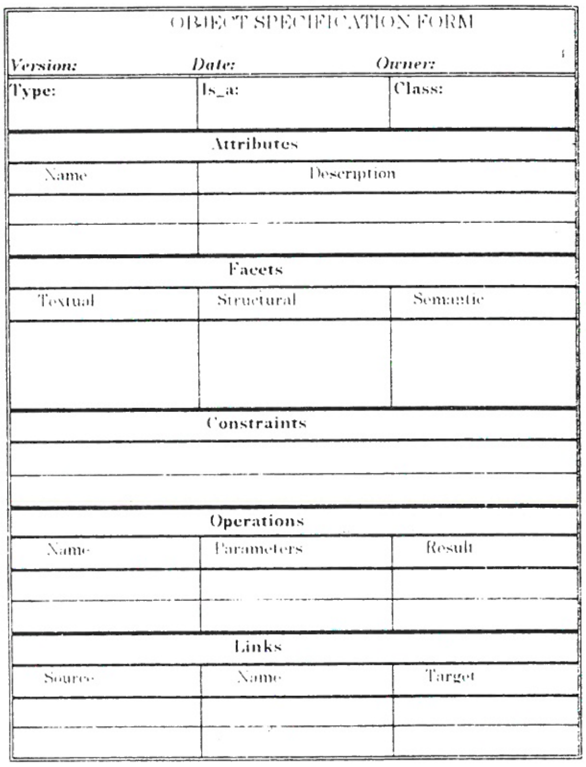


Figure 3. Conceptual Relationships in Object Modelling

Interobject *links* are binary relationships, that is sets of object pairs: an object for each role of the relationship. They specify how the objects of one type could be logically related to objects of another type. More formally we consider that the link is anchored in the first of the two types and could be specified by: its name, the linked object type and the cardinality of each role in the binary relationship. The cardinality of a role means the minimal and maximal number of occurrences that an object belonging to the role type could accept- in the link pairs. Expressions as: "0,*" (any number of objects are linked), "0,1" (at most one object is linked), etc. could be used for specifying the cardinality. Often, for an identified relationship its inverse relationship exists, too.
Figure 3 summarizes the various conceptual relationships concerning an object type.

2.1.5. Object Specification

In the early stages of analysis all the information about objects is to be gathered in *object forms* like those in Figure 4. Most sections in this form are optional. Initially, some of them are to be filled with sentences in natural language. Then, they will be gradually formalized and refined. All subsequent versions must be preserved. During analysis similar objects should be gathered in classes. As in the OO paradigm, a class specifies a collection of objects with the same structure and behavior. Classes are specified in *type forms* much similar to the form in Figure 4. When a new class is introduced most of the object forms addressing the objects into the class could be abandoned. Classes could be further classified in superclasses and so on. The class hierarchy is represented by the **is_a** fields of the type forms. The structure and services of an object are those specified in the type form of its class plus those inherited from the class superclasses. A CASE tool for classes and versions management is obviously assumed.
Facet fields address another hierarchy. They link objects with their subobjects. As already stated, the facets in the object's cluster are conceptually bound together by their belonging to the object definition. As a consequence, along object-subobject links each subobject inherits constraints and/or updating procedures from the originating facet. They should be added to the **Constraints** section of the subobject form.



OBJECT SPECIFICATION FORM

Version: *Date:* *Owner:*

Type:	**Is_a:**	**Class:**
Attributes		
Name	Description	
Facets		
Textual	Structural	Semantic
Constraints		
Operations		
Name	Parameters	Result
Links		
Source	Name	Target

Figure 4. Object Type Specification Form

In the **Links** section the objects related to the object are listed. On the left side (**Source** list) the objects that are anchors in their relationships with the object are included. On the right side (**Target** list) the objects that are in relationship with the object but are not anchors are included.
The **Operations** section includes the services the object should carry out. Formally, services are functions with parameters and result. The parameters can be external (events or "collaborators" which the object is related with by links) or internal (attributes, facets). The result could be a simple value, an attribute, a facet or an event.

2.2. System Activities

In the following the activity entity concept is examined. In our method the term "activity" is given a very general acceptance, including connotations of other terms as action, operation, or process. An activity *is doing* something easily identifiable as a notable event in the system life: state transformation, I/O actions, monitoring or controlling signals, etc. An activity is useless without one or more agents able to execute it. It and its agents enter the same relationship as a process and its potential processors do.
An activity is embedded in an *environment,* which is an activity in itself, specialized in service support and resource allocation for all the activities using it. To accomplish its task an activity may use other activities. Thus, there are two activity hierarchies: a functional one and another one of resource allocation and service support. Each activity has its space of states, mainly consisting of the objects it uses. Any activity may communicate with the others in the same environment to either synchronize their activities or transfer data thanks to a communication mechanism provided by the environment.
Any activity has a dual aspect: descriptive and operational; it is first of all planned, described, and then, executed. Even if a proper description lacked before activity execution, a rough sketch of it no doubt took shape in the agent's mind. That is why the two sides of the coin always exist: the activity description and the processes carrying out that description. So, we can talk about an activity type and its instances. This is precisely the classification relationship that exists between an object type and an instance of it. The analyst should identify the system activities, classify them into types, and specify them in an implementation-free manner.
Our activity model has two parts: a static one and a dynamic one. The static part encapsulates attributes (name, access rights, environment, etc.), agent types, used tools, and objects the activity uses or modifies.

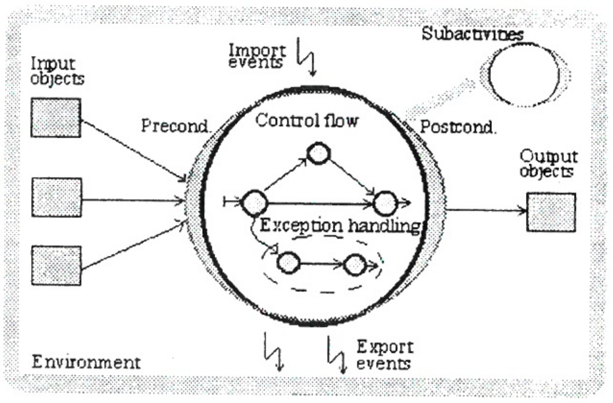
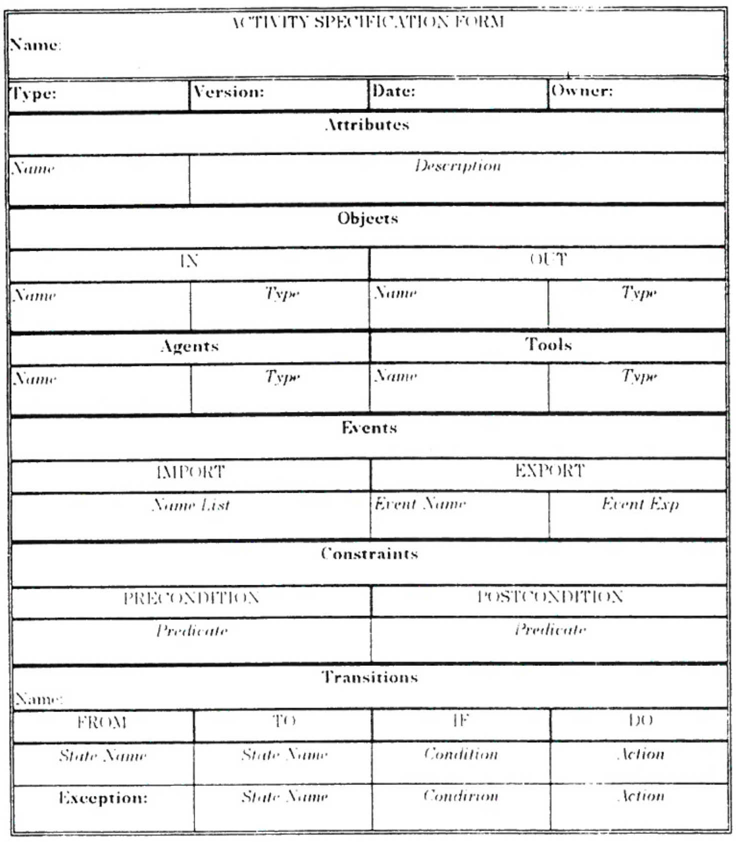


Figure 5. A Hint of the Activity Concept

The dynamic part includes aspects related to the activity execution: control flow, dynamic constraints, and exception handling specification (if any).
Figure 5 presents a hint of our activity entity concept. An activity processes some input objects and outputs other ones. To carry out an activity, a *precondition* must hold. After activity execution a *postcondition* holds, too. Activity execution may be motivated by the occurrence of some external *events* and during its execution events arising from the activity could also occur.
A form for activity specification is presented in Figure 6. Initially, it is likely that the form sections are filled with phrases in natural language. Then, a more formalized language could be gradually introduced, until a complete formalization is achieved. In this form one could insert graphical representations as DFDs, STDs,

PNs aimed to render the specification more readable. These representations could also comment on the language that accompanies the method. In any way the analyst should come to an agreement with the user on the formal tools for expressing analysis information.



ACTIVITY SPECIFICATION FORM

Name:			
Type:	Version:	Date:	Owner:
Attributes			
Name	*Description*		
Objects			
IN		OUT	
Name	*Type*	*Name*	*Type*
Agents		**Tools**	
Name	*Type*	*Name*	*Type*
Events			
IMPORT		EXPORT	
Name List		*Event Name*	*Event Exp*
Constraints			
PRECONDITION		POSTCONDITION	
Predicate		*Predicate*	
Transitions			
Name:			
FROM	TO	IF	DO
State Name	*State Name*	*Condition*	*Action*
Exception:	*State Name*	*Condition*	*Action*

Figure 6. Activity Specification Form

To specify the control flow in activities, a finite-state machine (FSM) approach was chosen: the user should identify some stable states in the activity progress and describe all possible transitions between these states. As Figure 6 shows, there are four fields for specifying transitions: the source and target states, an IF condition and a DO action. Transitions are enabled only if the IF clause holds. The IF clause tests objects belonging to process state and can include **when** event-based expressions. If a transition is allowed, subactivities and operations specified in the **Do** action are executed. To exit an activity a target state has to be reached. We consider activities as transactions, that is the objects involved in an activity are modified only if it succeeds. If during their execution something wrong or unexpected happens the **Exception** section is activated and out of several actions only one is selected. The first **If** clause that holds selects the action to be executed.

3. System Analysis Method

3.1. The method strategy

In the following the analysis method steps are briefly covered. Examples are drawn out from the requirements specification of a security system of the protected area of a nuclear power plant. A description of the system is found in Appendix A. Other, more complex examples are presented in [3].

The method strategy is quite simple:

The analyst should try to better know the application as the user sees it. An effort is then necessary for transferring the user's application universe in the method terms,while identifying and describing the main components of the model. Finally, an iterative refinement process completes the analysis.

With the delivered specification one could go on designing and then implementing it with some conventional methodology (for instance, with the structured design method [4]). But there is some other way, perhaps more promising, to future research. Because of its high conceptual level of approach, the final specification is suitable to being transformed into an object-oriented specification [5, 6]. We tested this way and the result was satisfactory. Moreover, important elements of the conventional, process-oriented methods are present in our analysis method. They could be benefited for obtaining a mixed specification, perhaps the best solution for many application problems. Many graphical tools used by structured methods (DFD, ERD, STD, etc.) could also be used to make the information contained in various sections of our specification language more readable. Redundancy (if any) is not critical in an analysis.

3.2. Identifying Events

Our claim is that an event-driven approach similar to that made in [7] is the best one in the early stage of analysis. From our interviews with users, we have noticed that the scenario for the future system's behavior is always centred upon event occurrences and the manner the system must respond to various events ("When I query ... the computer should display ...", "When an alarm occurs the video matrix must couple automatically the camera ... with the monitor ... and with the video sequence ...", "Each day, at 00h:00m:00s the system configuration should be updated", "Peripheral devices should be monitored at a rate of max. 500 msec.", and so on). Rough narrations about what has the system to do and when should it be accepted as first schemata of the system specification.

3.3. Identifying Main Components

The analyst conducts the user questionnaire for explicitly identifying in his or her scenario the agents and their main activities and for collecting them. As result, it is likely that external systems emerge and communicate with the system as well as with the main functions of the system. It is also likely that the main objects emerge as entities that are used or modified by activities. A linguistic analysis of the narration in order to identify subjects, actions, and objects of actions is useful in defining the main entities.

A good idea is to make the user view the upper conceptual level of the system behavior as a finite state machine where the machine states are the main activities of the system. This metamodel of behavior is useful in analysis because it is very general and within anybody's reach . On this territory users and analysts may have a great opportunity for converging their points of view. Such a diagram is presented in Figure 7. The activities resulted from the user's view focussed on (and guided) system states or functioning regimes:

- dialog state, when a user's command or query is expected;
- alarm state, when an alarm event has been identified and the system must display the map and the intervention procedure, couple a camera with a monitor and eventually a video sequence, and drive a relay or an action sequence;
- failure state, when a device failure was declared and the system must react similarly as in the alarm state;
- programming state, when the communication with the peripheral devices is off-line and the system configuration is modified, with some of its components, etc. changed.

Each state is viewed as an activity that the system executes until an event ends the execution (for instance, the alarm approach by the user puts an end to the alarm activity) or brings it into a new state for the execution of another, more urgent activity (for instance, an alarm or failure event brings the system from the Dialog activity, the default activity of the system, into the Alarm or Failure activity). While the Dialog and Programming have a single instance (the system is mono-user!) which is suspended when other activities enter the execution, all other activities could have more instances at a time (more alarms, more failures, or more queries to databases). Figure 7 captures this functioning.

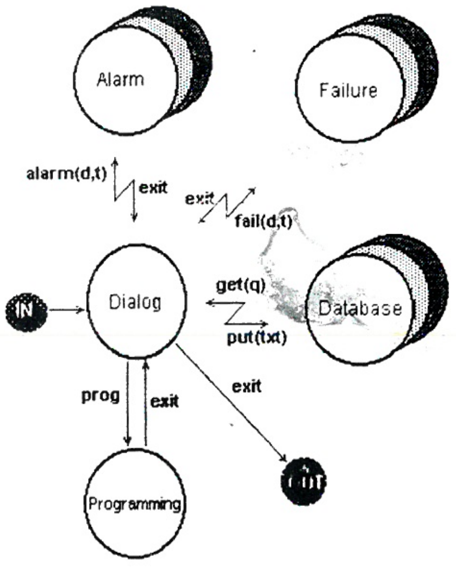


Figure 7. System Main States (Activities)

An initial ER-like diagram using the relationships shown in Figure 1 and involving the identified entities must also be drawn. The external entities should be marked on this diagram.
Before leaving this step a re-analysis of all entities derived from the initial, rough scenario is necessary for once for all fixing their nature. This task is not simple because, as already stated, the nature of entities involved in a system is fluctuating with the conceptual level the user approaches the system. Tuning the right approaching level for identifying the nature of system entities is the most demanding task the analyst must face at this step of analysis.
Often, in complex systems, there are activities that support and manage several subactivities supplying them with their local resources (objects, tools, agents). Such an activity that we called *algebra* [3] behaves in respect to its own objects and activities as a large "umbrella" protects several people isolating them from external interferences. An algebra is an environment where several activities evolve around some clustered or closely related objects. The concept is useful for modeling large phases or subprocesses in the system's global activity. In algebras activities could be easily combined in larger processes evolving sequentially or concurrently. Considering each algebra as a large environment supporting objects and activities is a good idea with a view at decoupling the system main components.

3.4. Specifying Entities

Within the general framework presented so far, a formalism is introduced to more accurately specify the details. Such a formalism should address the concepts the method uses. In the following, some hints about facets and activities specification are given. To illustrate our approach, specifications are provided in Appendices B and C for an object and an activity.
Although analysts have their taste preferring to come up with either the static parts of the application universe (the objects or data) or the dynamic ones (the activities or processes), it is our opinion that the specification should evolve concurrently at the same conceptual level for all components. Conserving the same level of abstraction until its complete analysis is over is a condition of coherence in the approach and a test of the problem understanding. This tactics is valid for the upper levels. Then, at the lower levels, an in-depth work could be a good idea especially for critical components.

3. 4.1 Object Specification

To specify object facets, the analysis method should provide a language or a graphical representation for describing:

- high- level object structures such as: set, list, table, tree, or graph,
- navigation mechanisms for each structure type, and
- primitive basic objects such as: integers, strings, symbols, but also simple objects belonging to the application universe without a relevant internal structure.

An effort should be considered for carrying out the object types. Supertypes may be introduced when object types manifest similarities. However, such generalizations are not compulsory in this stage of analysis because they might obscure some useful aspects to be revealed during analysis.
Appendix B shows an object type, SensorGroup. It represents objects that are made up from simpler objects of the Sensor type. The sensors in a sensor group are viewed either acting as a whole in a single set or acting as two sets: connected and disconnected sensors. These were the only facets identified in the user scenario for the system behavior.

3.4.2 Activity Specification

Beside some features common to all specification languages, we considered the following primitive components for activity specification:

- *Basic operations.* represent application-oriented object transformations without an internal, relevant to modelling purposes structure;
- *Support operations* belong to some general mechanisms supplied by the environment activity for: object manipulation (**create**, **delete**, **fetch**, **store**), activity activation (**execute**, **wait**, **cancel**), activity communication and synchronization (**send**, **receive**), exception handling (**save**, **restore**), event processing (**when**), user interface operation (**read**, **write**, **query**, **report**);

- *Operators* permit specification of expressions and control structure. Among them, the most important ones are: logical and arithmetic operators for expression building, and sequence, decision, selection and iteration operators for control structuring.

In Appendix C the Alarm activity is presented. Alarm has an input parameter: the object that signalled the alarm event. The temporal co-ordinates of the alarm event occurrence are stored in the **Attributes** section. During Alarm evolution instances of some subactivities are created and executed. They are listed in the **Subactivities** section. The **Transitions** section introduces two transitions: EXITING and ACTIVATION, and three states: INIT, EXIT, WAITING. The first two are predefined. Transitions are enabled when the user enters DEL and ENTER commands (here, simple key strokes) (Figure 8). During transitions, instances of new activities are created and executed, and others are made expire.

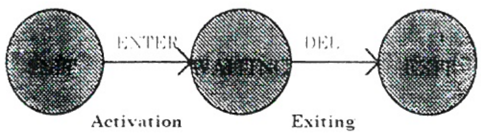


Figure 8. State Diagram of the Alarm Activity

3.5. Adding New Components

The initial scenario must be repeatedly confronted with already identified components. Each time the scenario is "re-played" at lower conceptual levels the need for introducing new components (new roles in the system!) is likely to occur. For instance, in order to describe how the Alarm activity will process a video sequence (a cyclic list of monitor-TVcamera couplings) or an action sequence (a list of relay switching actions) when an alarm occurs, two new activities could be introduced: TVCouple Management and Output Management. TVCouple Management is an activity that expresses what happens when a request for a video sequence arrives. Firstly, an instance of the TVCoupleManagement is created and activated. This instance controls a cyclic sequence until a request for ending the activity emerges (usually when the alarm processing having generated the initial request ceases its activity). An instance of the same activity could be re-used in other activities, for instance in a Dialog activity when the user can produce a command of coupling a monitor to a video sequence. Such refinement operations should be repeatedly executed until simple, unstructured actions are obtained. A similar approach is valid for objects.

4. Concluding Remarks

The paper has examined a conceptual approach to system analysis meant for making the requirements specification more understandable from the user's point of view.

The approach identifies in any system four categories of entities: objects, activities, agents, and tools. The paper has concentrated upon objects and activities. The conceptual view of these entities was advocated. A formalism for their specification was proposed.

Although the paper style is pragmatic, much work was invested in laying a theoretical foundation of this approach. More about this topic is found in [3]. Much further research is needed, too. This research should:

- elaborate new criteria for classifying the system components in the four categories;
- consolidate the set of primitives to be used in specification;
- derive global information on the system behavior from the system requirements specification;
- clarify the relationships of the method with other analysis methods (or methodologies), given that a mixture of models, techniques, and formalisms originating in various methods is the option of most today analysts.

REFERENCES

1. WIRFS-BROCK, R.J. and JOHNSON, R.E., **Surveying Current Research in Object-Oriented Design**, COMM. ACM, 33, 9, September 1990.

2. FICHMAN, R.G. and KEMERER, C.F., **Object-Oriented and Conventional Analysis and Design Methodologies**, COMPUTER, 25, 10, October 1992.

3. SERBANATI, L.D., **Integrating Tools for Software Development**, YOURDON PRESS, Prentice Hall Building, Englewood Cliffs, NJ, 1993.

4. YOURDON, E., **Modern Structured Analysis**, Prentice Hall, Englewood Cliffs, 1989

5. **Systems Application Architecture, Common User Access: Advanced Interface Design Guide**, IBM Corp., Doc. no. SY0328-300-R00-1089, 1989.

6. **OOA Tool**, Object International, Inc., Austin, TXS, 1991.

7. COAD, P. and YOURDON, E., **Object-Oriented Analysis**, YOURDON PRESS, Prentice Hall Building, Englewood Cliffs, NJ, 1991.

8. WARD, P.T. and MELLOR, S.J., **Structured Development of Real-Time Systems**, YOURDON PRESS, Englewood Cliffs, N.J., 1985.

9. HENDERSON-SELLERS, E. and EDWARDS, J.M., **The Object-Oriented Systems Life Cycle**, COMM. ACM 33, 9 , September 1990.

BRAS System

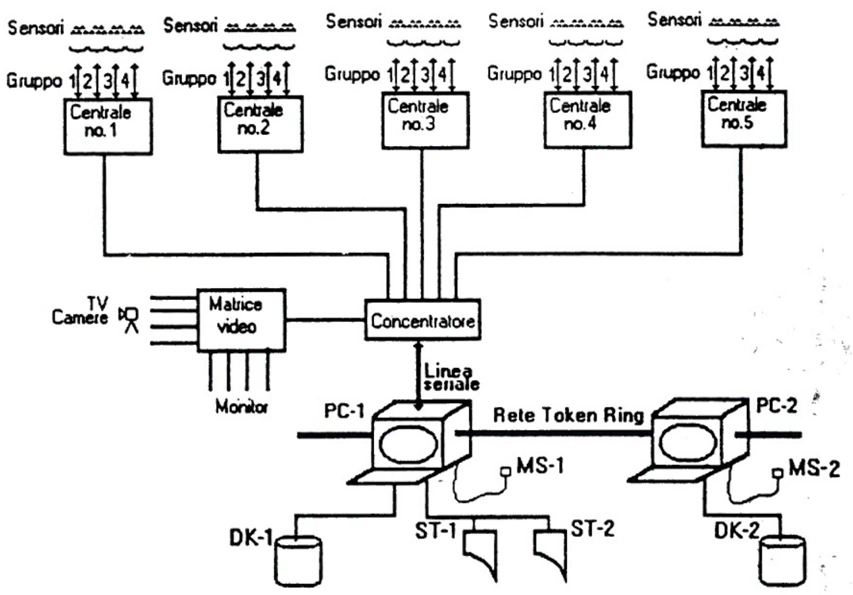


Figure 9. Architecture of the BRAS System

Figure 9 shows the hardware configuration of the system BRAS as it was imposed by the user. The system uses two networked PCs. The server in the network (PC-2) is dedicated to technological events processing. PC-1 manages all other events, the communication with the I/O devices and the database. There are some peripheral devices (specialized concentrators) able to manage locally I/O devices: sensors and relays. A multiplexer facilitates the dialog between the PC and peripheral devices. A TV switching station allows coupling between any TV camera and any monitor.

The system functions are:

- automatic scheduling of the activation of sensors, sensor groups, areas, I/O registers, relays, TV cameras, and monitors. The time schedule for any device is week-oriented. The days in a week may be differently scheduled from each other. The daily time schedule of any device provides time intervals for device activation;
- manual control by means of commands sent to system components;
- detection of such events as:
- – intrusion in secured areas,
- surpassing of the threshold values of technological (logical and analogical) parameters,
- – commands and queries supplied to the system by the responding personnel,
- – failure of the peripheral equipment (sensor, sensor groups, registers,relays, and CCTV);

providing an immediate reaction to any event:

- – sounds an alarm signal,
- – displays on a TV monitor a scene or more scenes,
- – in a cyclic sequence ,
- – displays area of the monitored site where the event occurred, displays all the events

occurred in the meantime and waits for them being considered by the guards,
- displays intervention procedure,
- executes one or more actions (a relay closure or writing in an output register),
- prints all information about the event,
- records in a database all information about the event;

- continuously monitoring all sensors and sensor groups checking for proper ooeration or for any failure occurrence;
- check for any shift of guards;
- diversified, password-based access to the system's facilities;
- automatic saving of the configuration;
- fully or partially saving of the event database;
- report and statistics generation;
- programming the system configuration components:

- number of hardware components,
- alarms (alarm = sensor + alarm type + alarm description + map + intervention procedure + camera-monitor couple or video sequence + action-output device couple or action sequence),
- sensor groups (group = set of sensors physically connected to a peripheral device and acting as a unit),
- areas (area = set of sensors occasionally joined for acting as a unit),
- week time schedules (list of daily schedules),
- daily time schedules (list of activation or deactivation moments during a day).

APPENDIX B

Sensor Object Specification

```
Shell SensorGroup
Alias "Sensor group"                                      // wording for the object type name
Is_a Object                                               // the super-type
Class SensorGroups                                        // the name of the object set
Attributes
 {
 Levels
  Alias    "Levels of the grouping tree" ? natural        // a natural value interactively updated
           value ( 0 ) ;                                  // default value
 MaxCompNr
  Alias    "Maximal number of components in the group" ? natural
           value ( 0 ) ;
 Type
  Alias "Concentrator type" ? text
           value ( "" ) ;
 }
Facets
{
 allinset :
   {
   Textual :   listbox ( ( for all x in $ insert (x)) )    // the facet is represented as a listbox
   Structural :  set_of Sensor ;                           // the facet is viewed as  a sensor set
   }
  comp:
   {
    Structural : record_of {                              //  group = connected  or disconnected sensors
                          connect : set_of Sensor ;
                          disconnect : set_of Sensor ;
                                 }
   }
}
Links
 {
 (1, 1) Gr_Schedule (0, *) WeekSchedule ;                  // Gr_Schedule =  link between a group and a
                                                            week schedule
 }
Operations
 {
 AddSensor
  Alias "Add new sensor" : ($, Sensor) -> $ ;              // $ is the current object
 CompNumber
  Alias "Get component numbers" ($) -> natural ;
 ElimSensor
  Alias "Eliminate sensor" : ($, Sensor) -> $ ;
 Monitor
  Alias "Monitor the group functioning" : ($) -> $ ;
 SelectSensor
  Alias "Sensor selector" : ($) -> Sensor ;
 }
Constraints
 {
 cardinal ( $[allinset] ) <= MaxCompNr ;                   // the sensor number is limited to MaxCompNr
 $[comp].connect } $[comp].disconnect == VOID              // the comp components are disjoints
 $[comp].connect | $[comp].disconnect == $[allinset]       // the jointed comp components = allinset
 }
Comments
{ A sensor group is a set of sensors acting as a whole when an enabling/disabling command is received }
End_Shell
```

APPENDIX C

Alarm Activity Specification

```
Activity Alarm
Env OnLineEnv                                              // on-line environment
Input
{
 inp : InputDevice ;                                       // alarmed sensor
}
Attributes
{
 occurr_time : time ;
 occurr_date : date ;
}
Agents CentralPointComputer, PeripheralComputer ;          // computers able to carry out the activity
Subactivities DisplayAlarm, TVCouplingManagement, OutputManagement ;
Private
{
  vs, as : ObjectVar ;                                     // object variables
  video, action, disp : ActivityVar ;                      // activity variables
}
Import key_stroke ;                                        // imported event
States WAITING ;
Transitions
{
  EXITING:
     From WAITING
        When key_stroke;
        If key == KEY_DEL ;                                // the operator cancels the alarm
        Do
          {
           cancel (video);
           cancel (action);
           cancel (disp);
          }
        To EXIT ;

    ACTIVATION:
       From INIT
         When key_stroke;
         If key == KEY_ENTER ;                             // the operator acknowledges the alarm
          Do
            {
             disp = create (DisplayAlarm) ;
             disp (input_device_to_map (inp), inp->Location ) ;   // displays the alarm map on the screen
             if ( exists (vs=video_schedule (inp)))               // Is there a video processing of the alarm?
               {
                 video = create (TVCouplingManagement) ;
                 video ( vs);                                     // a monitor with a TV camera is coupled
               }
             if (  exists (as=action_schedule (inp)) )            // Is there an automatic reaction?
               {
                action = create (OutputManagement) ;
                action ( as );                                    // some relays are switched
               }
            }
          To WAITING ;
}
End_Activity
```