

# An Object-oriented Rule-Based Approach to the Dynamic Modelling of Information Systems<sup>1</sup>

**A. Tsalgatidou**

EDP Department, Greek P.T.T.  
afrodite@di.uoa.gr

**P. Loucopoulos**

Department of Computation, UMIST  
pl@sna.co.umist.ac.uk

## Abstract

The problems inherent in the task of developing and maintaining large scale information systems have received wide coverage. The essential problem with these systems is their inflexibility to change, compounded by the fact that most contemporary software processes fail to adequately address the specific problem of system evolution.

This paper argues that the development of information systems requires the adoption of a new paradigm which recognises the explicit separation of organisational policy from programming code. To this end the paper discusses a rule-based model which is used for the specification of all control aspects of an information system. The definition of this model is achieved within a framework which is based on an object-oriented approach. This framework requires the definition of objects on the basis of a binary relationship model and the identification of operations on these objects. The invocation of these operations is handled by the rule-model.

Rules pertaining to the behaviour aspects of an object class may be viewed in terms of a network graphical notation. Such a network can be used for the dual purpose of *factoring* a rule-base as well as *animating* the behaviour of each object class.

## 1. Introduction

In recent years there has been a growing realisation that the development of large information systems is becoming increasingly more difficult as user requirements become broader and more sophisticated. Consequently, the area of requirements specification is receiving particular attention by developers since it is becoming obvious that improvements in this area are likely to yield considerable advantages in both quality of software and productivity of project personnel [5]. A major criticism levelled at most traditional requirements specification approaches is their poor handling of the capturing and modelling of the knowledge of the universe of discourse [14, 3, 31]. These shortcomings can be attributed partly to the inherent nature of the *process* of capturing, modelling and verifying concepts of the modelled domain and partly to inadequate *formalisms* used for the representation of these concepts.

Requirements specification implies two basic activities: modelling and analysis [11]. Modelling refers to the mapping of real world phenomena onto basic concepts of a requirements specification language. These basic concepts serve as the means of building various structures which, in their totality, constitute

---

<sup>1</sup> In *Proceedings of CAISE-90, Stockholm, Sweden, May 1990, Lecture Notes in Computer Science, 436, G. Goos & J. Hartmanis (eds.)*, Springer-Verlag, pp. 251-264.

the requirements specification for a problem domain. Analysis refers to techniques used to facilitate communication between requirements engineers and end-users making use of the requirements specification as the basis of that communication.

Because one of the primary uses of a requirements specification is in understanding a specific application domain the models used in expressing the specification must be *cognitive* in nature. These models, generally referred to as conceptual models, must be relevant to the milieu in which the information system is used, (i.e. the object system), and not related to its design or implementation. Moreover, the models should force active participation of users by stimulating and generating questions as to how reality is abstracted and assumptions are made.

A number of desirable properties for a requirements specification (defined in the form of a conceptual schema) have been proposed [4, 34, 32, 24] and can be summarised as follows.

**Implementation Independence.** No implementation aspects such as data representation, physical data organisation and access, as well as aspects of particular external user representation, (such as message formats, data structures, etc) should be included in a conceptual model.

**Abstraction.** Only general aspects of an information system and the universe of discourse should be represented (i.e. those not subject to frequent change). Abstraction results in a schema in which certain details are deliberately omitted.

**Formality.** Formality implies that descriptions should be stated in an unambiguous syntax which can be understood and analysed by a suitable processor. The formalism should be based upon a rich semantic theory that allows a clear relationship between descriptions in the formalism and the world being modelled [21].

**Constructiveness.** A conceptual model should be constructed in such a way as to enable easy communication between analysts and users and should accommodate the handling of large sets of facts. In addition, such a model needs to overcome the problem of complexity in the problem domain, by following appropriate abstraction mechanisms which permit decomposition in a natural manner.

**Ease of Analysis.** A specification needs to be analysed in order to determine whether it is ambiguous, incomplete, or inconsistent. A specification is ambiguous if more than one interpretation can be attached to a particular part of the specification. Completeness and consistency require the existence of criteria against which the specification can be tested. However, the task of testing for completeness and consistency is extremely difficult, normally because no other specification exists against which it can be tested [25].

**Traceability.** Traceability refers to the ability to cross-reference elements of a schema with corresponding elements in a design specification and ultimately with the implementation of an information system.

**Executability.** The importance of executability is in the validation of a specification [3, 15, 18]. In particular it refers to the ability of a specification to be simulated against relevant facts in the modelled reality. The executability of the descriptions in a schema is subject to the employed formalism.

These requirements for a conceptual model lead to two key questions about the modelling approach that should be followed:

- *what to model*, i.e. what aspects of the information system and the universe of discourse need to be captured in a particular specification?
- *how to model*, i.e. what method should a developer follow for constructing an appropriate specification?

The first question is addressed in this paper from the perspective of organisational policy. The argument put forward, which is also supported by others c.f. [13, 19], is that one of the most important factors towards the development of information systems which are capable to react to changes in their environment is the explicit modelling of application domain *policy*. In other words, it is proposed that in order to address the problem of system evolution successfully, there is a need to seek solutions which identify areas of change and explicitly represent all volatile elements of the application domain. On the basis of this premise the paper discusses a rule-based model which is used for the specification of all change-related (i.e. control) aspects of an information system.

The second question is addressed in this paper from an object-oriented perspective. An information base is considered as consisting of a set of objects which have certain behaviour. The identification of these objects follows a linguistic approach which determines entities and their relationships. These conceptual relationships are based on Fillmore's case grammar [12] and this approach has proved successful in the fields of database design [23] and knowledge representation [29, 17]. On the basis of this approach, first a binary relationship model is derived and subsequently this model is used to derive a set of object classes. The behaviour of each object class is defined in terms of operations whose control is determined by the rule-based model.

The importance of defining the control aspects of an information system in declarative terms cannot be overestimated. It is the authors' premise that these control aspects reflect evolution within the system's environment which in turn is guided by domain specific policy (from strategic to operational levels). The authors argue that in order to address the problem of system evolution successfully, developers must be provided with a process which identifies areas of potential change and offers explicit representation of these *volatile concepts*. This approach conforms to the recommendations in [3] and provides the means of maintaining specifications at a high level of abstraction by separating the goals of a system from the functions which realise these goals. If such a separation is not achieved, as is the case with most contemporary approaches, then those very elements which determine change in the system are specified and maintained only in programming code. The implication of this approach is that realisation of evolution in information systems requires considerable effort with unpredictable results due to the difficulty in locating and altering the appropriate code which represents the policy to be changed. To emphasise this point consider for example, a system which was examined by Anderson et al [1], which takes a set of clock timings for employees as input and produces a set of pairs  $\langle no-of-hours, rate-of-pay \rangle$ . Rates applicable to an individual are determined by policy relating to the individual, the department and the company. Whilst this is a relatively simple problem, in terms of business policy (only 8 business rules are involved), it requires a program of over 3,500 lines of code to deal with it. The approach proposed in this paper would define under what conditions the operations are to be executed and ignore all aspects of control flow.

The paper is organised as follows. Section 2 discusses the framework within which a rule-based specification may be developed. This framework is established along the lines advocated by the object-oriented paradigm. To this end, the model and techniques for defining objects and operations on objects are respectively described in sections 2.2 and 2.3. The approach is demonstrated using examples from the case study of appendix A.

On the basis of the proposed framework, the specification of the behaviour of all operations (and by extension the behaviour of the entire information system) is determined in a rule-based paradigm. Such an approach to specifying evolutionary aspects of systems is the subject matter of section 3. Section 3 introduces the syntax and semantics of dynamic rules. The concepts discussed in section 3 are explained using again the case study of appendix A.

Section 4 introduces a graphical notation which is used for *grouping* dynamic rules according to their effect on an object class and is also used as the basis of *animating* the behaviour of each rule by presenting a model with plausible triggering conditions. The grouping aspect serves as the means of factoring a potentially large rule base whereas animation assists in the validation of the model by acting as the means of communication between systems analysts and end users.

## 2. An Object-Oriented Framework: Objects and Operations

### 2.1 *Advantages of an Object-Oriented Approach*

Object-oriented development is an approach based on the concept of developing a software system in terms of objects and their interactions. The object-oriented philosophy has gained many advocates in the programming field, (see for example [20, 8, 22, 28, 9, 6]). In this section, the basic principles underlying the object-oriented philosophy are applied to the requirements specification area.

In an object-oriented framework the basic unit of decomposition is the object. An object, which can be a class or an instance of a class has certain behaviour and is associated with certain functions. An object is regarded as an autonomous entity, responsible for its own behaviour. It suffers actions and can trigger actions on other objects, in order to perform some of its functions. The actions performed by an object, are called the interface of the object.

One important advantage of using an object-oriented framework at the requirements specification level is that such a framework comes close to a human's perception of the real world. As Borgida et al [7] point out, "*the chief advantage of object-oriented frameworks is that they make possible a direct and natural correspondence between the world and its model*". Consider the wholesaler company example in Appendix A. In this example, an object-oriented view about the way this company operates would be to consider the company as a collection of various objects which belong to object classes (e.g. COMPANY, PRODUCT, CUSTOMER etc.). These objects interact with each other and exhibit certain behaviour. For example, the main behaviour of a COMPANY object is to check the stock of each product every end of month, to produce a product list at the end of every week and to take care of the requests that are made by customers for certain products. In order to achieve all these tasks, the object COMPANY interacts with other appropriate objects, i.e. PRODUCT, CUSTOMER, etc. When it is time for the company to check the stock of all products, the COMPANY object interacts with all the instances of the object class PRODUCT, each one of which checks its own stock.

Another advantage of the object-oriented framework is that it addresses information hiding, abstraction and inheritance. Information hiding, refers to the principle of hiding design decisions about the abstractions that are being employed to the model of reality. Abstraction, when used during system development, refers to the process of emphasising some aspects of a system, which are considered important, while suppressing others. Inheritance refers to the ability of objects to inherit properties and behaviour of other objects. Inheritance is one of the important and useful features of object-oriented frameworks as it helps to avoid repetitions in system descriptions.

### 2.2 *Definition of Object Classes*

A key characteristic of the requirements specification area is that the project activities involve much informality and uncertainty [33]. Consequently, the definition of objects appropriate to the modelled domain is far from a trivial task. Most object-oriented approaches make the assumption that somehow a set of object classes are identifiable. The paradigm developed in this paper derives an initial set of objects on the basis of a linguistic approach. This approach is based on the premise that most of the information about an application domain is conveyed to developers in the form of statements (during interviewing of key personnel in the application domain) as well as from examination of different documentation (forms, reports etc) relevant to the application.

Therefore, a conceptual schema is viewed as a set of individual *elementary facts*. An elementary fact is a true elementary proposition. For example,

- 1a. "John works for Accounts"
- 1b. "Accounts employ John"

References to individual things in the universe of discourse is made through labels. Examples of labels taken from the above set of elementary propositions are *John* and *Accounts*. A thing of the universe of

discourse which is referenced by a label is known as an entity. An entity class is an aggregate of entities for example, PERSON, DEPARTMENT, SUBJECT. The example elementary proposition 1(a) can now be defined as:

- 1a. Person with name 'John'  
works for  
Department with department name 'Accounts'

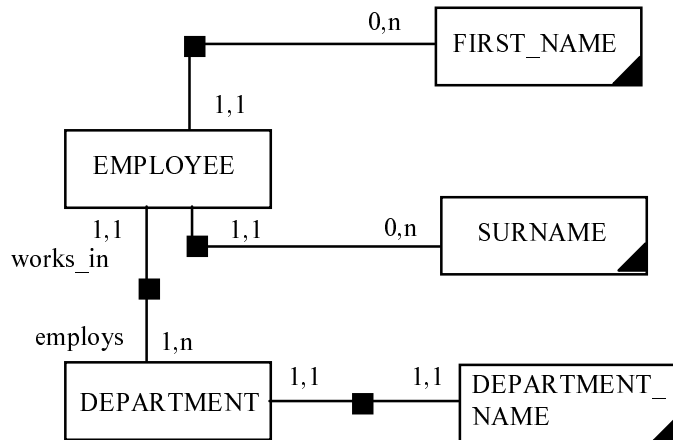
This proposition shows two types of relationship. One type associates entities to labels, known as a *reference* relationship, and the other type is confined to associating entities, known as a *fact*. An example of the former is 'Person has Name' where PERSON is an entity class and Name is a label class. An example of the latter is 'Person works for Department' indicating that both PERSON and DEPARTMENT are entity classes.

Each relationship class (reference or fact) represents one deep structure for which there may be two or more surface structures (depending on the number of things that the relationship associates). Each surface structure represents the *role* (also called *sentence predicate*) that an entity class or a label class plays in the relationship. For example, 1(a) and 1(b) below together represent a single, deep structure i.e. a relationship class between two entity classes.

- 1a. Employees work in Departments
- 1b. Departments employ Employees

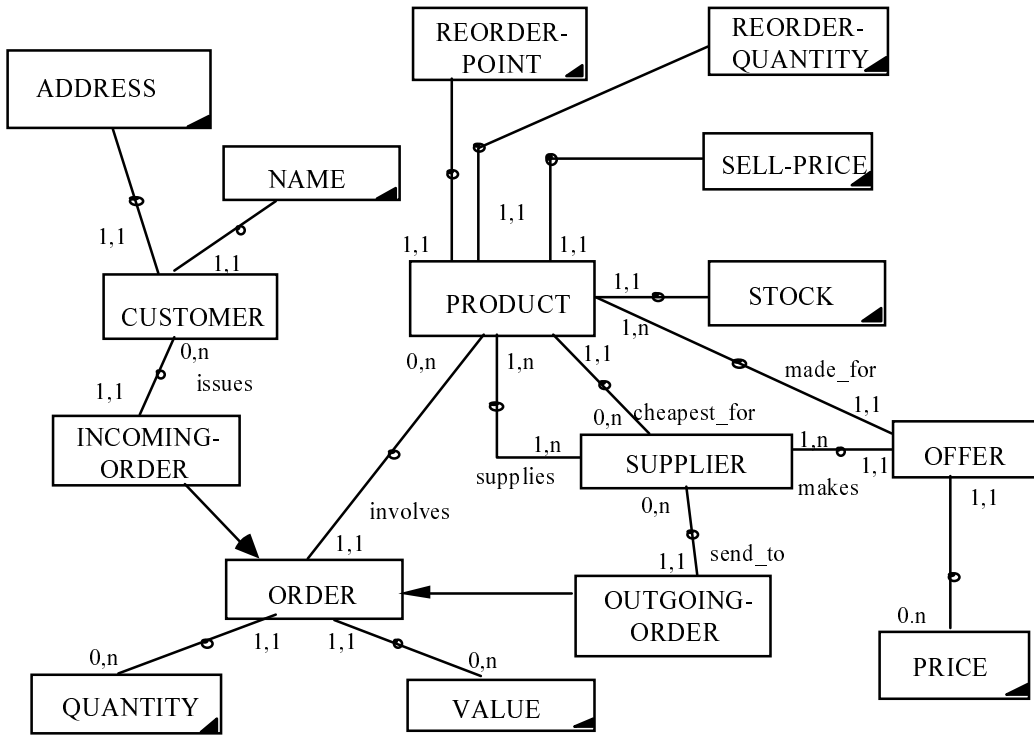
The sentence predicate 1(a) represents the role that the entity class EMPLOYEE plays in its association with the entity class DEPARTMENT. The sentence predicate 1(b) represents the inverse.

A graphical notation of an integrated set of elementary facts in the form of an information diagram, of which an example is shown in figure 1, can be used to accommodate better communication between developers and users. The information diagram depicts a number of entity classes, label classes and sentence predicates and the cardinality constraints which exist in each set of relationships. For example, in figure 1 it is shown that 'a DEPARTMENT has at least 1 and at most 1 DEPARTMENT\_NAME' where DEPARTMENT is an entity class and DEPARTMENT\_NAME a label class.



**Figure 1.** Example of an Information Diagram

On the basis of this analysis, an information diagram for the wholesaler company described in appendix A is shown in figure 2.



**Figure 2.** Information Diagram for the Wholesaler Case Study

In practice, the process of deriving an information diagram would involve much iteration due to the need to refine the schema.

On the basis of the information diagram, a set of object classes can be derived. Each object class corresponds to an entity class which has as its properties all references and selected fact classes. For example, from figure 2 the object **PRODUCT** can be defined as follows:

```

defclass PRODUCT
slots    stock, type integer(0, 10000),
          reorder_point, type integer(0, 10000),
          reorder_quantity, type integer(0,10000),
          sell_price, type integer(0,10000),
          sold_by, type instance_of(SUPPLIER),
          cheapest_supplier, type instance_of( SUPPLIER),
endclass.

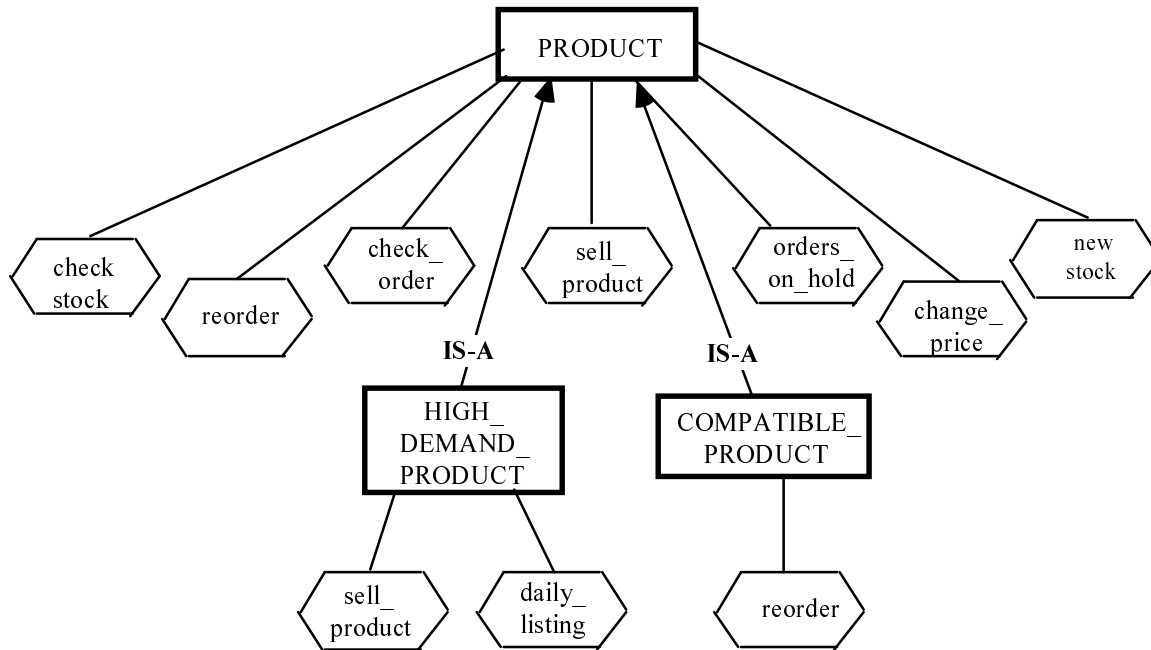
```

### 2.3 Definition of Object Behaviour

The definition of object classes represents the specification of the elements of the static aspects of an information system. Equally important is the specification of the dynamic aspects. This is achieved through the definition of the behaviour of each object class.

The behaviour of an object class is organised into small separate units according to their intended use. The importance of these units, known as *operations*, lies in the partition of the behaviour of an object and in the grouping of dynamic rules (discussed in section 3) which determine the control aspects of the behaviour of each object class.

An operation describes a function of a system or a particular situation where the object of interest plays an important part. An object, if it suffers actions or acts upon other objects, will be associated with one or more operations. For example the operations associated with the object class **PRODUCT** are: *check stock*, *reorder*, *sell the product*, *change price*, *produce stock list* and *new stock*. A special case may occur where an object may be active without having any operation attached to it. This is the case where the behaviour of an object is described in the operation of another object. Consider for example the object class **ORDER**. The behaviour of this class may be described in the operations of other object classes for instance, **PRODUCT** and **CUSTOMER**. Obviously, the allocation of an operation to an object class is a matter of choice for the developer. The attachment of operations to object classes may be shown graphically, in an *object behaviour diagram*, an example of which is shown in figure 3.



**Figure 3.** An Example Object Behaviour Diagram

The object behaviour diagram of figure 3 shows the object class **PRODUCT** and its operations together with its subclasses and their respective operations. The semantics of specialisation is as follows. A specialised object class is a class which inherits the characteristics and the behaviour of its parent class. In general, a specialised class inherits the operations of its parent class. However, two special cases may exist:

- A subclass responds to events which do not trigger the generalised class. This case is denoted by giving a different name to the operation of the subclass. For example, the operation *daily\_listing* attached to **HIGH\_DEMAND\_PRODUCT** signifies the fact that it only applies to this object class and not its parent class.
- An operation of a subclass has the same goal as an operation of its parent class but its detailed functionality is different. In this case the operation has the same name as its corresponding operation of the parent class. However, the specialised operation overrides its generalised equivalent. For example, **COMPATIBLE\_PRODUCT** which is a specialisation of **PRODUCT**, inherits all the operations of its parent class except the operation *reorder* which is overridden by its own *reorder* operation.

The detailed specification of an operation represents the primitive actions to be carried out on an instance of the object class to which the operation is attached. For example, the object **ORDER\_LINE** may have operations, 'create', 'delete', and 'issue'. The operation for 'create' might be specified as follows:

1. ORDER\_LINE <- create (P)
2. S := any good SUPPLIER for PRODUCT.P.
3. PURCHASE\_ORDER\_LINE := self <- new;
4. QUANTITY of it := QUANTITY to be reordered for PRODUCT .P;
5. PRICE charged for it := PRICE at which a PRODUCT.P is supplied by SUPPLIER.S;
6. PRODUCT referred to on it := PRODUCT .P;
7. PURCHASE\_ORDER that contains it := S <- get\_purchase\_order.
8. DB <- add (OL).

This example is a fairly detailed definition of how an order line may be created. Similar definitions have been defined in the context of the RUBRIC project [31]. Line 1 contains a header, indicating the entity (*purchase\_order\_line*), the operation name (*create*) and a single parameter. Line 2 selects a good supplier for the product referenced by P. Line 3 creates a new purchase order line, with lines 4 to 7 assigning values to it. Line 8 finally adds the order line to a database object known as DB.

Notice that within the operation, other operations can be fired, such as on lines 3, 7 and 8. The operation message on line 7 also returns a value which is assigned to *purchase\_order that contains it* (the *ORDER\_LINE*).

### 3. Specification of Behaviour in Terms of Rules

An Information System can be viewed as an artifact with certain functionality defined by a set of actions and with certain behaviour which is the result of the execution of various actions which have been triggered by some agents. These triggering agents may be generated either by the environment or by the various parts of the Information System. For example, if the Information System is a stock control system, a happening in the real world, such as the arrival of new stock for a product, affects the Information System by triggering the actions that perform the updating of the stock quantity of the received product. Obviously, the system responds only to certain happenings of the real world. Furthermore, the execution of some actions in one part of an Information System, could affect another part of it and trigger some other actions at that part. For example, referring again to a stock control system, the result of checking the stock quantity of a product, could affect the actions in the part of the system that takes care of the reordering of low stock products. Therefore, the dynamic part of an Information System could be considered as consisting of small parts with predefined functionality. The functionality of all these parts is the functionality of the Information System as a whole. This functionality is described in a set of actions which are executed when they are invoked either by a happening in the environment or as a result of actions performed in other parts of the Information System. The grouping of these actions is carried out in terms of the concept of operation as defined in section 2.

In order to describe the behaviour of a system, one needs to use concepts that describe the interaction between various parts of the system and the interaction between the system and the environment. In this paper, these interactions are described using the concept of *signal*. Signals are messages which are generated as a result of the execution of some actions in the Information System or by the environment as a result of a happening in the real world. They convey information about what has happened and can trigger a set of actions in a part of the Information System. For example, if a customer's order has arrived, the environment will generate the signal *order\_arrived* which will trigger a number of appropriate actions in the Information System. Signals are also generated inside the Information System and they are the means of communication between various parts of the system itself. For example, the condition that the quantity of stock for a product has fallen below the reorder level will be a signal to order new stock.

Following this discussion it is possible to view the evolutionary aspects of an information system as a collection of actions which are executed whenever an appropriate signal is received. Therefore,

a *signal* is defined as a boolean expression which when evaluated to true invokes a set of actions that change the state of the information system. A signal may be generated in the environment of the information system or within the system itself.

The concept of signal by itself, however, is not sufficient for specifying the control aspects of a system's behaviour. This is achieved through the use of a *dynamic action rule* which serves as the means of specifying the coordination of the execution of all operations. Therefore, in the context of this paper,

a *rule* is the means of specifying the coordination of signals and operations, i.e. it defines the conditions necessary for the invocation of operations. A rule has the syntax:

```
"WHEN" <signal >
["IF" <precondition>]
"THEN" <action_part>.
```

```
signal:= env_signal | clock_condition | state_condition | state_operation.
precondition:= state_condition | clock_condition.
action_part:= operation_message.
operation_message:= object_class_name [variable] "<-" op_name [parameter].
```

The WHEN part of a rule is compulsory. It implies that *when the triggering condition becomes true* the rule may be considered further and in this sense the WHEN clause is considered as the *temporal* control mechanism of the model.

A signal may be: a control message from the Information System's environment e.g. 'reorder request obtained'; or a condition on time periods and the system clock, e.g. 'end of the month'; or a condition expressed in terms of the definition of an object class, e.g. 'quantity\_in\_stock < quantity\_of\_reorder\_point'; or a database operation expressed in terms of an object class, e.g. 'before\_create\_order'.

The precondition part determines whether a rule may fire and may be expressed in terms of either conditions on the value of an object's property or in terms of a clock condition. Signals and preconditions may be conjunctive or disjunctive expressions. The following are example rules for the object class PRODUCT from the wholesaler case study in the appendix.

```
rule R1:  WHEN  SIGNAL order_arrived (INCOMING_ORDER.O)
           IF    O.CUSTOMER.PAYER_STATUS = bad
           THEN  CUSTOMER <- order_on_waiting_list (O, C)

rule R2:  WHEN  SIGNAL order_arrived (INCOMING_ORDER.O)
           IF    O.CUSTOMER.PAYER_STATUS = good
           THEN  PRODUCT <- check_order (O)

rule R3:  WHEN  SIGNAL before_create_orderline (INCOMING_ORDER.O, PRODUCT.P)
           IF    O.QUANTITY < P.STOCK AND
                P.STOCK > P.REORDER_POINT
           THEN  PRODUCT <- sell_product (O, P)

rule R4:  WHEN  SIGNAL before_create_orderline (INCOMING_ORDER.O, PRODUCT.P)
           IF    has_failed (rule_R3)
           THEN  PRODUCT <- order_on_hold (O, P)

rule R5:  WHEN  SIGNAL process_orders_on_hold (INCOMING_ORDER.O)
           THEN  PRODUCT <- check_order (O)

rule R6:  WHEN  SIGNAL new_stock (PRODUCT.P, QUANTITY.Q, SUPPLIER.P)
           THEN  PRODUCT <- new_stock (P, Q, S)
```

In this sample set of rules, rules R1, R2, R5 and R6 have a control message from the environment as their signal. Such a signal would normally be generated by the user of the system through some dialogue

screen. Rules R3 and R4 have a different invocation strategy since they have an internal signal in the form of a state operation. Observe that both these rules have as signals the state operation *before\_create\_orderline*. This state operation is in fact generated as part of the operation *check\_order* but this is transparent to both of these two rules. Furthermore, R3 and R4 are totally unconcerned about the detailed actions of the *check\_order* operation and as far as their triggering part is concerned the necessary signal may be generated at any part of the information system and these rules will always respond in the same way. Thus, details about *how* some control aspects of the information system are to be executed are of no concern at the specification level but emphasis is instead placed on *what* needs to be carried out.

The example rules R1-R6 are meant to apply to the object class PRODUCT. As discussed in section 2.3 any object class may be specialised into other classes. In the example case study, PRODUCT is specialised into HIGH\_DEMAND\_PRODUCT and COMPATIBLE\_PRODUCT. This specialisation implies not only differences in the structural component of the subclasses but also different behaviour. According to the description of the business rules for the wholesaler case study (see appendix A), for a high demand product, i.e. of type A, B or C, whenever the stock is below the reorder point for these products, good customers will have their order immediately processed. Therefore, the behaviour of HIGH\_DEMAND\_PRODUCT will be different to that of PRODUCT. Specifically, rules 3 will be overridden by rules R3.1 and R4.1 respectively:

**Rule 3.1:** WHEN SIGNAL **before\_create\_orderline** (INCOMING\_ORDER.O, PRODUCT.P)  
IF P is\_a HIGH\_DEMAND\_PRODUCT AND  
O.CUSTOMER.PAYER\_STATUS = good  
THEN PRODUCT <- sell\_product (O, P)

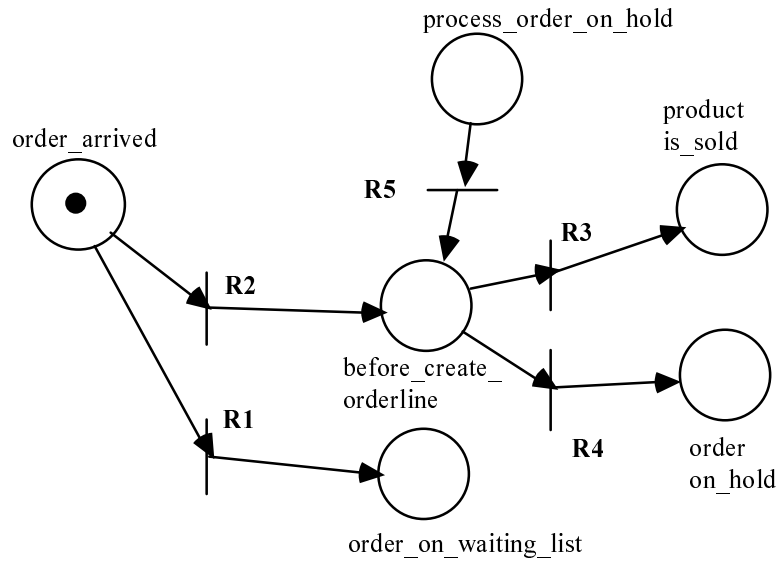
**Rule 4.1:** WHEN SIGNAL **before\_create\_orderline** (INCOMING\_ORDER.O, PRODUCT.P)  
IF P is\_a HIGH\_DEMAND\_PRODUCT AND  
O.CUSTOMER.PAYER\_STATUS = bad  
THEN PRODUCT <- order\_on\_hold (O, P)

Summarising the methodological aspects of the proposed paradigm, a developer would define a set of object classes pertinent to the application domain, the basis of some linguistic analysis and with the assistance of an information diagram. For each object class a set of operations will be defined and each operation will be detailed in terms of the basic actions relevant to the attached object class. The invocation conditions i.e. control and coordination of a system's operations will be defined declaratively in terms of rules.

#### 4. Graphical Notation

Because of the potentially high volume of rules and the complexity of their interdependency, a specification expressed in terms of the constructs described in section 3 should be viewed at a higher level of abstraction. It is obvious that a textual representation of rules lacks adequate abstraction facilities. Therefore, a graphical notation which permits the modelling of the interaction of these rules is a desirable property of the proposed model. If in addition this notation exhibits formal properties then the specification would lend itself to *animation* which can be exploited for highlighting the behaviour of the system to potential users. This requirement leads to the consideration of the Petri Net formalism and its many extensions [27]. Of particular relevance to the requirement stated above is the Augmented Petri Net approach [35].

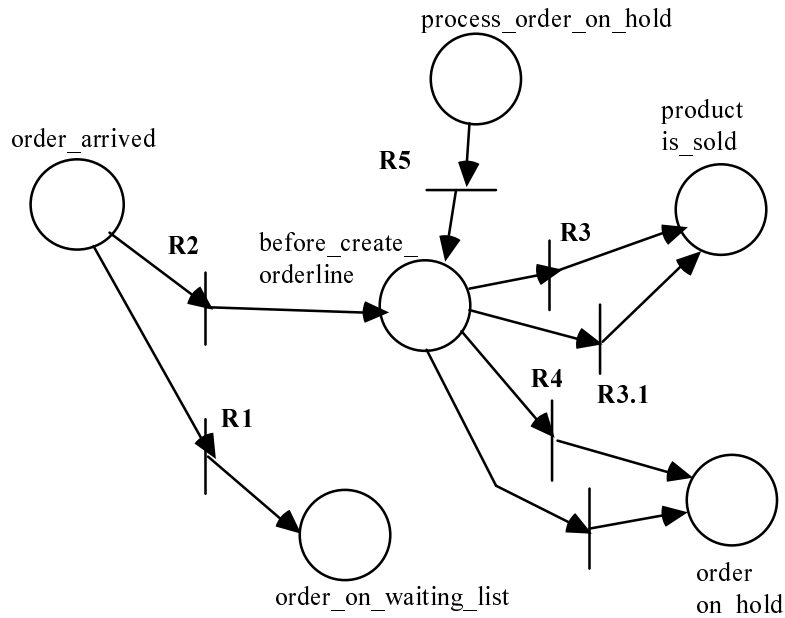
The graphical formalism introduced in this section uses *places* to represent signals (the WHEN part of rules) and *transitions* are inscribed with the IF and THEN parts. Since every rule needs a signal to be triggered, all the transitions will have at least one input place representing the triggering signal. An example net for the rules derived by considering the behaviour of PRODUCT presented in section 3 is shown in figure 4.



**Figure 4.** A Net for the Rules of PRODUCT

The dynamic behaviour of the model can be demonstrated by the firing of the net. When a place associated with a signal holds a token, this means that the signal is present and the rules inscribed to the transitions which have as input this place, are enabled. If the precondition part is satisfied, the rule fires and the marking of the net changes accordingly. The firing of the rules may generate other signals which can subsequently enable other transitions. Consider for example that a customer's order has arrived and the environment has generated the signal *order\_arrived*. The place associated with this signal would then receive a token. Rules R1 and R2 will then be marked as being enabled. If the precondition part of each one of these rules is satisfied, the action part of each rule will be executed and new conditions will then apply. For rules R2 and R5, the same operation will be invoked (*check\_order*) will try to create a new orderline (a database action). Before this database action takes place the triggering part of rules R3 and R4 will be enabled and one of these two rules will be executed.

There are two main advantages from organising and representing graphically rules according to the behaviour of each object class. The first advantage is a natural decomposition of what could potentially be a large and complicated set of rules. For example, consideration of the rules corresponding to the behaviour aspects of *HIGH\_DEMAND\_PRODUCT* would result in a net as shown in figure 5.



**Figure 5.** A Net for the Rules of HIGH\_DEMAND\_PRODUCT

The second major advantages of the net model is that, its graphical representation coupled to its formal semantics, lends itself to animation. A software system has been developed to edit and animate nets with places as signals and transitions inscribed by rules [30]. This graphical animation facility may be used for demonstrating the dynamic behaviour of the modelled system and for providing help in the verification of the rule-based specification. For example, animation of the model can help with detecting redundant and conflicting situations by highlighting the rules inscribed to every transition every time a transition is enabled. Redundancy occurs when two or more rules can fire in the same situation giving the same results whereas conflict occurs when rules that fire in the same situation produce contradictory results.

One of the model's features is that the number of tokens that each place can hold is countable. This feature enables the detection of circular rules by the assignment of output places to some transitions which would serve as counters of the number of times the transitions fire. A place assigned as output to a transition, receives a token every time this transition fires. If this place is not input to any other transition, its tokens will not be consumed. Therefore, the number of tokens that this output place will be holding will correspond to the number of times that the input transition has fired. If such a place is continuously receiving tokens, this may be an indication of the existence of circular rules.

Missing rules could also be detected by animation. For example, if a signal, which is expected to trigger some actions, has been generated and nothing is happening, this may mean that some rules are missing. Another indication of missing rules is when there are some transitions which never become enabled. If the triggering part of the rule is internal to the system, this would mean that the necessary rule for the production of the signal is missing.

## 5. Conclusions

Contemporary approaches to system development, whilst attempting to improve the quality of systems, have focused primarily on initial quality, with little attention given to the technical problems of system evolution. The result is that the software industry continues to produce inflexible systems which are difficult to maintain. The approach presented in this paper attempts to provide a better approach to building systems through the development of a software process and supporting tools which will explicitly accommodate those parts of a system that are regarded as volatile. Traditionally, ANSI/SPARC-style data management architectures and recent developments in HCI management tools

have enabled the separation of these elements of a system. The main initiative of the presented in this paper has been to extend this approach to include organisational policy. In particular, this approach seeks to separate out and explicitly maintain throughout the software lifecycle, the notion of policy, as described by and triggers and preconditions in dynamic rules. Complementary to this approach is work currently being carried out in the RUBRIC project [16] which enhances the paradigm presented in this paper by incorporating integrity rules at the specification level and by mapping all concepts from the specification level to the operational level. This theoretical work is supported by a software environment which assists system developers to first specify a system and subsequently develop and run applications.

The advantages of explicitly representing knowledge about a slice of reality are likely to be significant in the long term. It is the authors' belief that such an approach will lead to a number of benefits of which the most important are:

- better validation of captured requirements through the use of specification execution
- improved maintenance of an information system via the explicit representation and management of business policy (being analogous to data management)
- development of a specification in an incremental fashion, because of the separation between the what a specification is suppose to represent and how the functional requirements are realised.

The motivation behind this approach is the realisation that there is a need to provide a framework for the accurate representation of user requirements in a specification and to explicitly maintain this specification throughout the lifetime of an information system. To this end, the paradigm discussed in this paper seeks to provide a greater range of facilities than is currently practised for as Dubois et al [11] argue, contemporary specification languages while they provide concepts well suited to the description of algorithms, tend to be less appropriate to the description of application domains.

It is also becoming obvious that current conceptual modelling formalisms are oriented towards the functional specification of the software system rather than the definition of the problem domain [14]. If improvements are to be made in the quality of software then the knowledge about the application domain must be formalised and explicitly encoded. To this end, the work presented in this paper follows the premise that information system development is about formalising and documenting *knowledge* about the universe of discourse and this knowledge should be represented explicitly and independently to the way that it is implemented in data structures and algorithms, thus leading to a more efficient way of developing and maintaining software.

## References

1. Anderson, M. et al, "Report on Task A1: Research into the ability to use rules to describe the business and its activities", E928/R2/FINAL, James Martin Associates, Brussels.
2. ANSI, "Study Group on Database Management Systems: Interim Report", FDT, Bulletin of the ACM SIGMOD, Vol 7, No. 2, 1975.
3. Balzer, R. et al, "Software Technology in the 1990's: Using a New Paradigm", Computer, November 1983, pp. 39-45.
4. Balzer, R.M, and Goldman, N., "Principles of Good Software Specification and their Implications for Specification Languages", Proc. Spec. Reliable Software Conference, April 1979, pp. 125-128.
5. Boehm, B.W., "Software Engineering", in [10], pp. 97-121.
6. Booch, G., "Object-Oriented Development", IEEE Transactions on Software Engineering, Vol. SE 12, No. 2, February 1986.
7. Borgida, A. T., Greenspan, S., Mylopoulos, J., "Knowledge Representation as the Basis for Requirements Specification", Computer, Vol. 18., No. 4, April 1985, pp. 84.

8. Cohen, A.T., "Data abstraction, data encapsulation and object-oriented programming", SIGPLAN Notices, Vol. 19, No. 1, January 1984, pp. 31-35.
9. Cook, S., "Languages and object-oriented programming", Software Engineering Journal, March 1986, pp. 73-80.
10. Couger, J.D., Colter, M.A. & Knapp, R.W., "Advanced System Development/Feasibility Techniques", John Wiley & Sons, Inc., 1982.
11. Dubois, E., Hagelstein, J., Lahou, E., Ponsaert, F., Rifau, A., Williams, F., "The ERAE Model: A Case Study", in [26], pp. 87-105.
12. Fillmore, C. "The Case for Case", in Universals in Linguistic Theory, E. Bachs & R. Harms (Eds), Holt, Rinehart and Winston, New York, 1968.
13. Fjeldstad, R.K. et al, "Application program maintenance", in Parikh & Zveggintzov (1983) Tutorial on Software Maintenance, IEEE, pp.13-27.
14. Greenspan, S.J., "Requirements Modelling: A Knowledge Representation Approach to Software Requirements Definition", Technical Report No. CSRG-155, University of Toronto, 1984.
15. Karakostas, V. and Loucopoulos, P., "Verification of Conceptual Schemata Based on a Hybrid Object Oriented and Logic Paradigm", Journal of Information and Software Technology, Vol 30, No 10, December, 1988, pp. 587-594.
16. Loucopoulos, P., "The RUBRIC Project - Integrating E-R, Object and Rule-based Paradigms", Workshop session on Design Paradigms, European Conference on Object Oriented Programming (ECOOP), 10-13 July 1989, Nottingham, U.K.
17. Loucopoulos, P. and Champion, R.E.M, "Knowledge-based Support for Requirements Engineering", Journal of Information and Software Technology, Vol 31, No 3, April 1989, pp. 123-135.
18. Loucopoulos, P. and Karakostas, V., "Validating Conceptual Models of Office Information Systems", Software Engineering Journal, Vol 4, No 2, March 1989, pp. 87-94.
19. Mathur, R.N., "Methodology for Business System Development", IEEE Transactions on Software Engineering, Vol.SE-13, No.5, May 1987, pp.593-601.
20. Meyer, B., "Reusability: The Case for Object-Oriented Design", IEEE Software, March 1987, pp. 50-64.
21. Mylopoulos, J., "The Role of Knowledge Representation in the Development of Specifications" Information Processing 86, Kugler, H. J. (ed.) Elsevier Science Publishers B. V. IFIP 1986.
22. Nierstrasz, O.M., "An Object-Oriented System", in: Office Automation, D. Tschritzis (ed.), Springer-Verlag, 1985, pp. 167-190.
23. Nijssen, G.M., "On Experience with Large-scale Teaching and Use of Fact-based Conceptual Schemas in Industry and University", in Proc IFIP Conference on Data Semantics (DS-1), R. Meersman & T.B. Steel Jr (Eds), Elsevier North-Holland, Amsterdam, 1986.
24. Nijssen, G.M., Duke, D.J., Twine, S.M., "The Entity-Relationship Data Model Considered Harmful", Proc 6th Symposium on Empirical Foundations of Information and Software Sciences, Atlanta, Georgia, USA, October 1988.

25. Olivé, A., "Analysis of Conceptual and Logical Models in Information Systems Design Methodologies", In Information Systems Design Methodologies: A Feature Analysis, Olle, T., Sol, H., and Tully, C. (eds), North Holland Publ. Co. 1983.
26. Olle, T.W., Sol, H.G., Verrijn-Stuart, A.A. (eds.), "Information System Design Methodologies: improving the practice", North-Holland Publishing Company, IFIP 1986.
27. Petri, C.A., "Communication with Automata", Suppl. 1 to Tech. Rep. RAD C-TR-65-337, Vol.1, Griffiss Air Force Base, NY, 1966 (translated from "Kommunikation mit Automaten", University of Bonn, Germany 1962).
28. Rentsch, T., "Object-Oriented Programming", SIGPLAN Notices, February 1986, pp. 51-57.
29. Sowa, J.F., "Conceptual Structures: Information Processing in Mind and Machine", Addison-Wesley Publishing Company, 1984.
30. Tsalgatidou, A., "Dynamics of Information Systems: Modelling and Verification", Ph.D. thesis, Department of Computation, University of Manchester Institute of Science and Technology, June 1988.
31. Van Assche, F., Layzell, P.J., Loucopoulos, P., Speltincx, G., "Information Systems Development : A Rule-Based Approach", Journal of Knowledge Based Systems, September, 1988, pp. 227-234.
32. van Griethuysen, J.J. et al (eds), "Concepts and Terminology for the Conceptual Schema and the Information Base", Report ISO TC97/SCS/WG3, 1982, Publication No. ISO/TC97/SC5 - N 695.
33. Vitalari, N.P. and Dickson, G.W., "Problem Solving for Effective Systems Analysis: An Experimental Exploration", in Communications of the ACM, Vol. 26, No. 11, November 1983.
34. Yeh, R.T., "Requirements Analysis - A Management Perspective", Proc COMPSAC '82, pp. 410-416.
35. Zisman, M.D., "A Representation of Office Processes", Dept. of Decision Sciences, University of Pennsylvania, WP 76-1-03, 1976.

## **ACKNOWLEDGEMENTS**

The work reported in this paper has been partly funded by the Commission of the European Communities under the ESPRIT R&D Programme and the Greek PTT.

The authors wish to acknowledge the work of their colleagues in the RUBRIC project many aspects of which have influenced the work reported here. In particular the work of the following is gratefully acknowledged: Frans van Assche, Zissis Palaskas, Paul Layzell and Bill Karakostas.

## **APPENDIX A: Description of a Wholesaler Company**

Company B & S is a wholesale firm that runs its business in a simple and straightforward way.

### **A.1 General Domain Description**

1. From the supplier and customer organisation the name, the address and the contact person are known.
2. For a list of products there is information about the sales price, the quantity of stock in hand, the reorder point, the reorder quantity and the suppliers that can supply the product and at what price.
3. An order to a supplier (outgoing orders) consists of a list of order lines, each specifying the ordered products and quantity (price not mentioned).
4. We receive shipments from suppliers that match a set of order lines of the same supplier.
5. An order from a customer (incoming order) consists of a list of order lines, each specifying the ordered products and quantity.
6. A shipment to a customer matches a set of order lines of that customer.

### **A2. Business policies, decisions and rules**

At the time of implementation , the following policies were established:

1. At the end of the month, check the stock level of each product and if it is below the reorder point, order a quantity equal to the reorder quantity.
2. Buy the products from the supplier who sells it cheapest.
3. Every week a stock list must be produced.
4. Prices can only change overnight.

After problems that appeared because product A, bought from supplier X, was not completely compatible with product B, bought from supplier Y, the managing director decided that:

5. Product B can only be bought from suppliers that also sell product A and vice versa.

A year later the the company were selling a lot of A, B and C type products, so that the company run out of stock very often for these and only these products. Since sales had unpredictable peak sales, the managing director decided that:

6. For only products A, B and C, one should now reorder whenever the reorder point is reached and not wait until the end of month.
7. Whenever the stock is below the reorder point for these products (A,B,C only), good customers will have their order immediately processed. The other orders are put on hold until new stock arrives. Then, first come first served, applies.