

Specifying and Validating Requirements: The VENUS System

Aphrodite Tsalgaidou, Dimitris Gouscos, Constantin Halatsis

Department of Informatics, University of Athens, TYPA Buildings,
Panepistimioupolis, Ilisia, GR-157 71 Athens, GREECE
email: {afrodite, gouscos, halatsis} @ di.uoa.ariadne-t.gr

Abstract

Requirements specification and validation is recognized as a crucial part of the information systems development process. This paper presents the VENUS integrated environment for requirements engineering. The underlying conceptual framework of VENUS combines the entity-relationship, object-oriented and rule-based paradigms for data and behaviour modelling. VENUS tools enable systems analysts to formulate, analyse and validate requirements specifications which can be automatically mapped either to a C++ prototype or to the model of Rule-Based Nets (RBNs), a variant of Predicate-Transition Nets. RBNs constitute an executable model of requirements specifications and can be studied by analytical PN algorithms, animated and validated against test cases. The VENUS environment, therefore, provides an integrated, complete and formal, yet user-friendly, framework for requirements engineering. A VENUS prototype has been implemented so far, and work is still in progress towards (i) full-scale development and (ii) mapping of executable requirements specifications to design and implementation structures.

1. Introduction

The importance of requirements specifications in the information systems development process is not a recently identified need [Endres 75]. It has been earlier realised that incorrect requirements specifications result in the most serious errors in the information systems development process, because the cost to fix them increases as development progresses [Boehm 75]. However, there is still a lack of methods and tools to focus on the capture and validation of requirements specifications and the efforts reported in [Reubenstein 91], [Zave 91] and [Fickas 93] are among the few exceptions that can be found in this area. The large number of existing methodologies for systems analysis and design produce inherently static specifications. Furthermore, CASE tools play a passive rather than an active role in requirements specifications. In order to validate the captured requirements, developers try to derive system behaviour from static descriptions. What is needed is an approach and a set of tools that enables developers to obtain a working model of a system at the early requirements phase, so that its behaviour can be better understood and appreciated. This may be achieved if the produced requirements specifications are executable. An executable specification is a formal model of the system which can simulate the system's behaviour when executed by a suitable interpreter. Thus, such a specification can be thought of as a

prototype of a proposed information system and therefore it should be comprehensible, modifiable, easy to check for internal consistency and free from bias towards specific implementation strategies.

To this end, this paper presents the VENUS system which is an integrated environment offering a number of tools to support the requirements engineering process. The tools offered by VENUS assist a system analyst to elicitate, specify, analyse and validate executable requirements specifications within an object-oriented framework. Petri nets [Murata 89] are used as the underlying formalism for expressing object-oriented requirements specifications in a graphical manner. Animation and execution are some of the mechanisms used by VENUS for validating object-oriented executable requirements specifications.

The object-oriented approach has been found appropriate for developing requirements specifications in the VENUS system for the following reasons:

- it enables the uniform modelling of various real world objects in a direct and natural way, concealing any specific details inside them;
- it allows to model dynamic behaviour and interaction of objects in a high-level language, making specifications easy to comprehend by non-programmers;
- it provides useful abstractions, such as generalisation, aggregation and specialization, which facilitate extensions of the model and incorporation of new sorts of information and

also result in compact and robust requirements specifications;

- it provides executable specifications which can be used for prototyping and validation purposes and for implementation of the final system in an object-oriented language.

The rest of the paper is organised as follows: the next section discusses work related to this approach; section 3 presents the VENUS environment; section 4 describes the stepwise construction of requirements specifications expressed in the Object-oriented Rule-based Model (ORM) and its translation to a C++ prototype; section 5 shows the mapping of ORM to the graphical Petri net-based RBN model and gives a formal definition of RBNs; in section 6 the validation of ORM is discussed and finally in section 7 the conclusions are presented.

2. Related work

The development of VENUS relates to work on object-oriented development as well as in the areas of formal methods, executable specifications and testing, Petri nets and graphical animation.

The object-oriented approach has been used in systems analysis and development and examples may be found in [Booch 92], [Coad 91], [Coleman 92], [Rumbaugh 91] and [van Baelen 92]. Furthermore, examples of the use of rules in requirements specifications may be found in [D'Haenens 90] and [Loucopoulos 91]. The VENUS approach demonstrates the combination of both the object-oriented and the rule-based paradigms within a uniform framework integrating all system aspects.

Formal software development methods like Z [Spivey 87], OBJ [Goguen 87], VDM [Jones 86] and its extensions [Durr 92] receive much attention in academic environments, but have yet to be used in large-scale industrial projects. The main reason for this, apart from their strong need for computerized support tools, is that their strict mathematical background is not easily understood by the majority of systems analysts and end-users, thus making difficult the construction and validation of formal specifications [Plat 92]. Based on the premise that the entire information systems development process, and requirements engineering in particular, should be undertaken in a framework that does not compromise user-friendliness for formality, the VENUS approach uses formal notations mapped to graphical models, thus attempting to come up with formally correct, yet easily understandable, requirements specifications.

The importance of constructing executable specifications has been realised by researchers and developers within the software engineering community and a number of executable system models, executable

specification languages and tools contributing in this direction have appeared lately. For example PAISLey [Zave 91] is a language designed for real-time distributed systems and uses an operational approach for building a requirements specification, emphasizing the definition of processes as the building blocks of the system. VENUS differs from PAISLey in the sense that requirements specifications are object-oriented and objects, rather than processes, are the basic building blocks of the system model.

In the line of thought of [Kemmener 85], where it is stated that executable requirements specifications must be tested early in the development process to ensure that the specified system can be implemented and operate in an effective and efficient manner, VENUS enables systems analysts to test specifications right after requirements capture and well before design. Again like the proposal of [Kemmener 85], the VENUS environment offers two sorts of testing tools: (i) a rapid prototyping tool (C++ code generator) and (ii) a symbolic execution tool.

Symbolic execution in VENUS is accomplished by means of graphical animation and Petri nets. Animation, on the one hand, has proven to be a valuable validation technique; results reported in [Finkelstein 92], [Dahler 87] and [Shand 88] are among the many examples in this area. On the other hand, Petri nets constitute a powerful formalism [Murata 89] and have been used as a modelling tool in various applications. For example, in [Lakos 93] a language for object-oriented Petri nets (LOOPN) is used for modelling a door controller protocol. Other examples may be found in [Brinkkemper 90] and [Kappel 91], where extensions of Petri nets are used to reconcile the informal requirements engineering process to the more strictly-defined system development activities. The expressive power of Petri nets has led to the development of many analysis and animation tools like PROD [PROD 93], Cabernet [Pezz• 92], UltraSAN [Obal 93], POSES [POSES 92] etc. The majority of these tools, however, trade advanced functionality for treatment of specific classes of Petri nets, whereas a generic and customisable PN meta-tool would be of great practical value. VENUS introduces a rule-based extension of Petri nets for modelling and animating object-oriented requirements specifications for validation purposes, and offers a custom-made graphical editor/ animator.

3. The VENUS environment

The VENUS environment is depicted in fig. 1. This figure shows the tools provided by VENUS as well as their input and output models. The usual sequence of steps that may be followed by an analyst during the requirements development process is also shown in this

figure. The tools provided by the VENUS environment are:

- A syntax-directed textual editor for constructing textual Entity-Relationship Models (ERMs).
- A graphical editor for constructing graphical ERMs.
- A mapping tool which takes as input ERM specifications and produces specifications of an Object-oriented Static Model (OSM). OSM specifications are static since they are produced from static ERM specifications, thus OSM must be extended to incorporate the dynamic behaviour of the identified object classes; this is accomplished with the use of the next tool.
- A syntax-directed textual editor for adding dynamic aspects to OSM specifications and producing specifications of the Object-oriented Rule-based Model (ORM).
- A C++ generator, which takes as input ORM specifications and produces executable specifications in C++.
- A mapping tool which takes as input ORM specifications and produces specifications of the RBN model.
- A graphical editor/ animator for the RBN model. This tool may be used to edit and animate graphical RBN specifications.

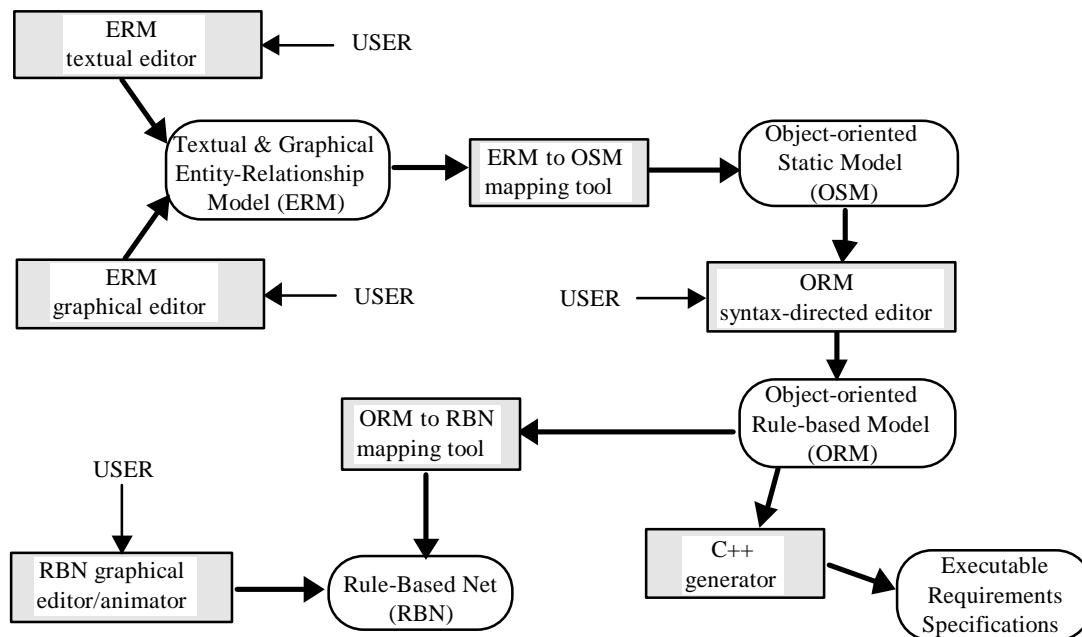


Fig. 1 The VENUS environment

Depending on the analyst's preferences and experience, the development of specifications may start at different stages of the VENUS development process. Furthermore, VENUS architecture allows more than one analysts to cooperate during the development process with each one of them being involved in a different stage.

In the following section we mainly concentrate on the object-oriented development of requirements.

4. The Object-oriented Rule-based Model (ORM)

Requirements specifications are formulated in an Object-oriented Rule-based Model (ORM) which contains the static and dynamic aspects of the various identified object classes. The behaviour of object classes is expressed in terms of Rules which are grouped in Behaviour Units (BUs). More specifically, the hierarchy of object classes may be constructed by first forming a binary Entity-Relationship Model (ERM) enriched with inheritance on the basis of a linguistic approach [Tsalgatidou 91]. Figure 2 shows an example of such a model.

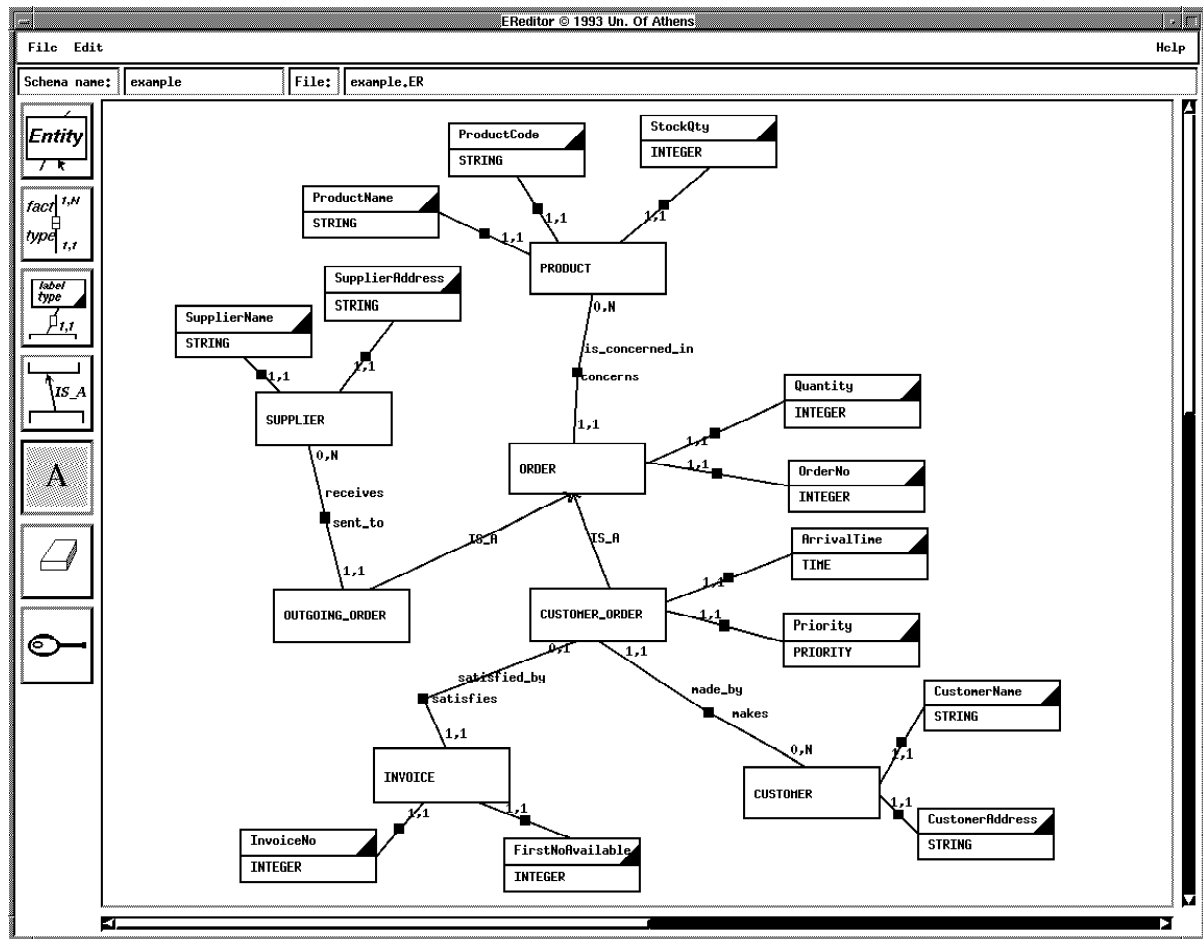


Fig. 2 An example of an ERM

The produced ERM is mapped to an Object-oriented Static Model (OSM) [Gouscos 92]. As is obvious, OSM contains only the specifications of the static system aspects. Therefore, OSM is subsequently enhanced so as to include the dynamic system aspects and any new object classes that may be needed. The ORM syntax-directed editor is used to incorporate these enhancements to OSM and produce the Object-oriented Rule-based Model (ORM). Fig. 3 contains part of the ORM which resulted from the ERM of figure 2. More specifically, fig. 3 shows the ORM specification which resulted from the mapping of the CUSTOMER_ORDER entity type of fig. 2 into the corresponding object class of OSM and from subsequent enhancement of this class with its dynamic aspects.

ORM models the dynamic aspects of object classes in their behavioural part using the concepts of *behaviour units (BUs)*, *signals* and *rules*. The static and behavioural parts of ORM complement each other, in the sense that the static part specifies only the

structural aspects of data without specifying any operations on them, while the behavioural part presupposes the existence of object classes and specifies their behaviour in terms of BUs, signals and dynamic rules.

The concept of *BUs* is used to partition the behaviour of an object class. Each BU belongs to one class of the object-oriented model and is associated with a specific triggering signal type. The receipt of individual signals of this type activates the behaviour described in this BU. A BU may be either *class BU* or *instance BU* and has the following general structure:

```
class | instance BU <BU_name>
triggered by <signal> (<signal_parameters>)
  <BU body>
end BU <BU_name>
```

The body of a BU is a set of dynamic rules having the form

```
[IF <preconditions> THEN] <actions>
```

The *preconditions* of a rule are expressed by a boolean formula and have to be satisfied before the *actions* described in its *THEN part* can be executed. The receipt of a triggering signal by a BU activates all the dynamic rules of that BU. The preconditions of rules of the same BU are mutually exclusive, so that exactly one of the rules will always fire. Actions in the *THEN part* of a rule may modify/create/delete object instances and/or produce some *signals* sent to other BUs or to the external environment of the modelled system. A rule belongs to exactly one BU and a BU belongs to exactly one class; therefore the dynamic behaviour of each object class is modelled as a collection of rules grouped in the BUs specified for that class and triggered by specific signals.

ORM supports simple inheritance of both static and dynamic aspects as follows:

- object classes inherit all the static properties of their superclasses and may introduce new ones
- object classes also inherit the BUs of their superclasses without any changes and/or redefine (some of) them by changing (some of) their existing rules or by adding new ones; they may also introduce new BUs describing behaviour not specified by their superclasses.

For each rule R, the rule model provides a *highest acceptor class* field (*hac(R)*) containing the name of the class where rule R is defined and a *lowest acceptor classes* field (*lac(R)*) containing the names of the lowest subclasses of *hac(R)* that still inherit rule R.

The dynamic behaviour of an information system is modelled as a sequence of firings of rules, grouped in BUs and triggered by specific signals. All the rules of a BU are triggered by the same signal. The firing of each rule, may result in the activation of other BUs by generating appropriate signals. The interaction of an

information system with its external environment is modelled by the receipt or sending of signals. Thus, signals serve as the means of communication between the various parts of an information system and between the system and its external environment. More information about ORM may be found in [Tsalgatidou 93].

The dynamic behaviour of a system modelled in this way can be demonstrated and checked by executing the system specifications. For this purpose VENUS offers a C++ generator tool, currently under development, which is used for transforming ORM specifications to C++ executable specifications. Appendix A depicts part of the executable specifications which are produced by applying the C++ generator tool to the ORM specifications of figure 3. In this way, developers can validate the captured requirements by executing the specifications and running test cases.

Apart from C++ executable specifications a graphical representation of ORM, accompanied with graphical animation and execution facilities and supported by formal tools, would provide greater help to its analysis and validation. Furthermore, the validation process would be even more facilitated by presenting the system's dynamic behaviour at different levels of abstraction. To this end, Petri nets are used in the VENUS environment as an underlying formalism to graphically represent the ORM model in a number of abstraction levels and to validate it by means of graphical animation and formal validation tools. The Petri net model used in VENUS is called Rule Based Net (RBN) and derived from ORM specifications using the ORM to RBN mapping tool (see fig. 1). In the following section we give a formal definition of the RBN model and we explain how ORM is mapped to RBN.

```
object class CUSTOMER_ORDER subclass of ORDER
instance properties
name ArrivalTime card 1,1 domain TIME
name Priority card 1,1 domain PRIORITY
name Customer card 1,1
  domain OBJECTID of CUSTOMER
name Invoice card 0,1
  domain OBJECTID of INVOICE
end instance properties
class behaviour
class BU CUSTOMER_ORDER.ProcessOrdersOnHold
triggered by ProcessOrdersOnHold(Product)
rule R1 hac CUSTOMER_ORDER
  lac {CUSTOMER_ORDER}
begin
execute NotifyOrders
  (signal.Product,OnHold) ;
end
end BU CUSTOMER_ORDER.ProcessOrdersOnHold
end class behaviour
```

```
instance behaviour
instance BU CUSTOMER_ORDER.ProcessNow
triggered by ProcessNow( )
rule R1 hac CUSTOMER_ORDER
  lac {CUSTOMER_ORDER}
begin
  if self.Quantity > self.Product.StockQuantity then
  execute SetPriority(OnHold) ;
  NoShipment(self.OrderNo) -> EXT ENV ;
end

rule R2 hac CUSTOMER_ORDER
  lac {CUSTOMER_ORDER}
begin
  if self.Quantity <= self.Product.StockQuantity then
  execute SetPriority(Processed) ;
  IssueInvoice(self.ObjectId) -> INVOICE ;
  DecreaseStock(self.Quantity) -> self.Product ;
end
end BU CUSTOMER_ORDER.ProcessNow
end instance behaviour
class procedures
procedure NotifyOrders
  (Product card 1,1 domain OBJECTID of PRODUCT,
  Priority card 1,1 domain PRIORITY)
begin
  for each Customer_Order where (Customer_Order.Product = signal.Product) and
  (Customer_Order.Priority = signal.Priority) do
  ProcessNow( ) -> Customer_Order ;
end
end class procedures
instance procedures
procedure SetPriority
  (NewPriority card 1,1 domain PRIORITY)
begin
  self.Priority := NewPriority ;
end
end instance procedures
end object class CUSTOMER_ORDER
```

Fig. 3 An ORM specification

5. The Rule-Based Net (RBN) Model

A Rule-Based Net is a Petri net-based model used for graphical representation of ORM specifications. RBNs resemble Predicate-Transition (PrT) nets [Genrich 81] augmented with additional information. Basically, an RBN contains inscribed places, inscribed transitions as well as inscribed arcs connecting places and transitions. An RBN also contains an underlying structure defining information used in the various inscriptions.

Analysts may interactively produce RBNs at different levels of abstraction using the ORM to RBN mapping tool. At the highest abstraction level, a “context RBN” is produced which contains only one transition for the behaviour of the entire modelled system, as well as input and output places for the signals sent to and from the external environment (see

fig. 4(a)). Places are inscribed with signal names and parameters of the corresponding signals. A “high-level RBN” is formed in the next abstraction level by decomposing the unique transition of the context RBN to one transition for each object class of ORM as well as to places for the signals exchanged between object classes (see fig. 4(b)). Subsequently, each object class transition of the high-level RBN may be further decomposed into other transitions and places, representing the BUs contained in the corresponding object class and the signals exchanged between these BUs. In this “medium-level RBN”, transitions model BUs and are inscribed with BU names, whereas places are again inscribed with the names and parameters of the corresponding signals (see fig. 4(c)). Each BU transition may be further decomposed into one transition for each rule contained in the corresponding BU and inscribed with the rule name. By following these steps for all the object classes the “low-level

RBN” is constructed which is the last but one abstraction level of RBN for the whole ORM (see fig. 4(d)). At the final step, the body of each rule (namely the *hac*, *lac*, *preconditions* and *actions* of each rule) is introduced into the corresponding rule transition thus producing a “detailed RBN” (see fig. 5(a)). This

detailed RBN is always accompanied by its structure Σ (see fig. 5(b)).

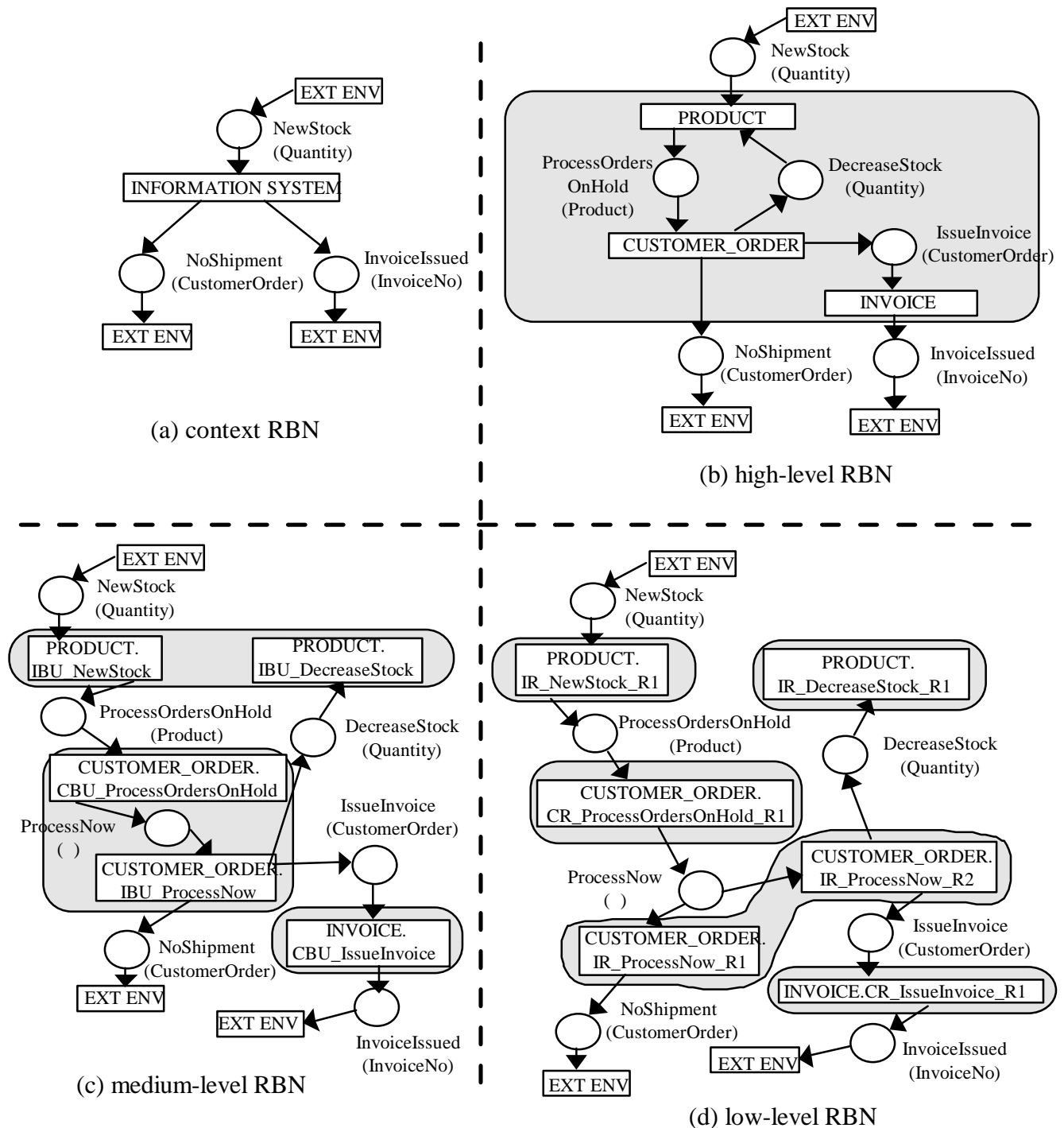


Fig. 4 Abstraction levels of the RBN corresponding to the behavioural component of an ORM partly depicted in fig. 3

A transition may produce more than one signals of the same type during one firing, as a result of the execution of the actions of the corresponding rule. The number of produced signals of a given signal type

denotes the multiplicity of the corresponding output arc.

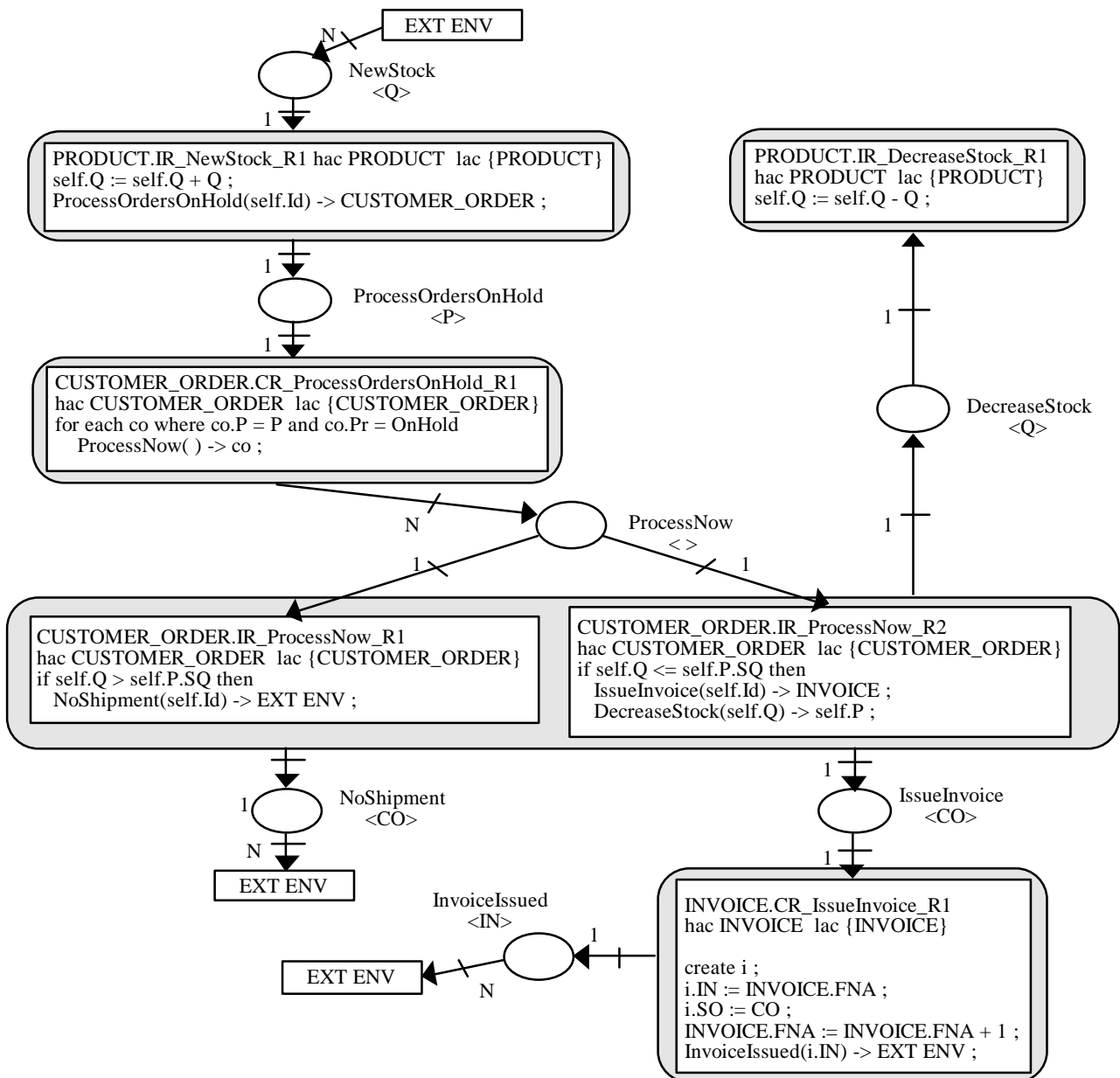


Fig. 5(a) Detailed RBN corresponding to the behavioral component of an ORM partly depicted in fig. 3

Thus, arcs are inscribed with an integer multiplicity. An arc from a place to a transition has always multiplicity equal to one, since a rule requires exactly one signal in order to execute. In the graphical representation of RBN, signal instances are depicted as

tokens. If a signal has no parameters, the corresponding token is represented graphically by an anonymous bullet; otherwise, the token is represented as a tuple of signal parameter values.

A detailed RBN is formally defined as a tuple

$R = \langle P, T, F, K, N, \Sigma, \text{sig}, \text{insc}, \text{hac}, \text{lac}, M_0 \rangle$

where

- P is a non-empty set of RBN places;
- T is a non-empty set of RBN transitions;
- $F \subseteq (P \times T) \cup (T \times P)$, with $F \neq \emptyset$, is a set whose elements are the RBN arcs;
- $K : P \rightarrow \{1, 2, \dots, \text{INF}\}$ is a function that maps each place of P to an integer number denoting the capacity of this place, i.e. the number of tokens that this place can hold; the value INF is used for some places with infinite capacity;
- $N : F \rightarrow N_{\text{def}} \cup N_{\text{undef}}$ is a function that maps each arc of F to an integer number denoting the multiplicity of this arc, i.e., the number of tokens transferred across this arc, each time the transition at the one end of the arc fires; the two multiplicity domains are defined as

$N_{\text{def}} = \{1, 2, \dots\}$, and

$N_{\text{undef}} = \{N_1, N_2, \dots\}$

with $N_i \notin N_{\text{def}}$

- Σ is the underlying structure of an RBN and has the form

$\Sigma = \langle D^{\text{val}}, D^{\text{obj}}, D^{\text{sig}} \rangle$

where D^{val} denotes the domain of signal parameters and of object instance properties, D^{obj} denotes the domain of objects of the RBN, and D^{sig} denotes the signal domain of the RBN (see fig. 5(b)).

- D^{val} represents the value domain of the RBN; D^{val} is a non-empty set of tuples of the form $\langle \text{IndivType}, \text{Relations}, \text{Operations}, \text{Values} \rangle$ Each of these tuples describes a specific type of values that may appear in the signal types that

label the places and arcs of the RBN, or in the properties of (the instances of) object classes.

- D^{obj} represents the object domain of the RBN, i.e. the collection of the object classes of the application. D^{obj} is a non-empty set of tuples of the form $\langle \text{ClassName}, \text{SuperClassName}, \text{Properties} \rangle$ For each object class of the object domain of the RBN, the set D^{obj} contains a corresponding tuple with the above three fields.
- D^{sig} represents the signal domain of an RBN; it is a non-empty set of signal types, i.e., a non-empty set of tuples of the form $\langle \text{SignalName}, \text{SignalParameters} \rangle$ Each of these tuples describes a specific type of signals that may appear as tokens in the RBN places.
- $\text{sig} : P \rightarrow D^{\text{sig}}$ is a function that maps each place of the RBN to a signal type from the signal domain of the RBN
- insc is a function which maps each transition of T to a dynamic rule
- hac is an element-valued function that relates each transition t of T to an object class called highest acceptor class of transition t - $\text{hac}(t)$ - which is the object class where the rule inscribed in transition t has been originally defined
- lac is a set-valued function that relates each transition t of T to some object classes called lower acceptor classes of t - $\text{lac}(t)$ - which are the lowest classes in the object schema that inherit the rule inscribed in transition t
- M_0 is an initial marking that assigns token to places.

1. Value classes (component D^{val})

$\{ \langle \text{OBJECTID}, \text{OBJECT}, \{=, \{ \}, \text{predefined} \rangle, \langle \text{STRING}, \text{OBJECT}, \{ \}, \{ \}, \text{predefined} \rangle, \langle \text{INTEGER}, \text{OBJECT}, \{ >, <= \}, \{ +, - \}, \text{predefined} \rangle, \langle \text{PRIORITY}, \text{OBJECT}, \{ =, \{ \}, \{ \text{OnHold}, \text{Immediate}, \text{Processed} \} \rangle, \langle \text{TIME}, \text{OBJECT}, \{ \}, \{ \}, \text{predefined} \rangle \}$

2. Object classes (component D^{obj})

$\{ \langle \text{OBJECT}, \text{OBJECT}, \{ \}, \{ \langle \text{ObjectId}, (1,1), \text{OBJECTID}, \text{all} \rangle \rangle, \langle \text{PRODUCT}, \text{OBJECT}, \{ \}, \{ \langle \text{ProductName}, (1,1), \text{STRING}, \text{all} \rangle, \langle \text{ProductCode}, (1,1), \text{STRING}, \text{all} \rangle, \langle \text{StockQuantity}, (1,1), \text{INTEGER}, [0-\dots] \rangle, \langle \text{OrderedIn}, (0,N), \text{OBJECTID of ORDER}, \text{all} \rangle \} \rangle, \langle \text{SUPPLIER}, \text{OBJECT}, \{ \}, \{ \} \}$

```

    { < SupplierName, (1,1), STRING, all > ,
      < SupplierAddress, (1,N), STRING, all > ,
      < PendingOrders, (0,N), OBJECTID of
        OUTGOING_ORDER, all > } > ,
  < CUSTOMER, OBJECT, { } ,
    { < CustomerName, (1,1), STRING, all > ,
      < CustomerAddress, (1,N), STRING, all > ,
      < OrdersMade, (0,N), OBJECTID of
        CUSTOMER_ORDER, all > } > ,
  < ORDER, OBJECT, { } ,
    { < OrderNo, (1,1), INTEGER, [1-...] > ,
      < Quantity, (1,1), INTEGER, [1-...] > ,
      < Product, (1,1), OBJECTID of PRODUCT, all >
    } > ,
  < CUSTOMER_ORDER, ORDER, { } ,
    { < ArrivalTime, (1,1), TIME, all > ,
    } > ,
  < Priority, (1,1), PRIORITY, all > ,
    { < Customer, (1,1), OBJECTID of CUSTOMER,
      all > ,
      < Invoice, (0,1) OBJECTID of INVOICE, all > }
  > ,
  < OUTGOING_ORDER, ORDER, { } ,
    { < Supplier, (1,1), OBJECTID of SUPPLIER, all >
    } > ,
  < INVOICE, OBJECT,
    { < FirstNoAvailable, (1,1), INTEGER, [1-...] > } ,
    { < InvoiceNo, (1,1), INTEGER, [1-...] > } > }

```

3. Signal types (component D^{sig})

```

{ < NewStock ,
  { < Quantity, (1,1), INTEGER, [1-...] > } ,
  { EXT ENV }, PRODUCT > ,
  < ProcessOrdersOnHold,
  { Product, (1,1), OBJECTID of PRODUCT, all > } ,
  { PRODUCT }, CUSTOMER_ORDER > ,
  < ProcessNow,
  { } ,
  { CUSTOMER_ORDER }, CUSTOMER_ORDER > ,
  < NoShipment,
  { < CustomerOrder, (1,1), OBJECTID of
    CUSTOMER_ORDER, all > } ,
  { CUSTOMER_ORDER }, EXT ENV > ,
  < DecreaseStock,
  { < Quantity, (1,1), INTEGER, all > } ,
  { CUSTOMER_ORDER }, PRODUCT > ,
  < IssueInvoice,
  { < CustomerOrder, (1,1), OBJECTID of
    CUSTOMER_ORDER, all > } ,
  { CUSTOMER_ORDER }, INVOICE > ,
  < InvoiceIssued,
  { < InvoiceNo, (1,1), INTEGER, [1-...] > } ,
  { INVOICE }, EXT ENV > }

```

Fig. 5(b) The structure Σ of the RBN of fig. 5(a)

Thus, an RBN is a simple formal model showing *control flow* within a system and at the same time hiding, in the corresponding transitions, information about *what* is happening in the system. One of the advantages of using RBNs for modelling the dynamic

behaviour of systems is that their graphical representation facilitates validation, as well as analytical study of system behaviour by exploiting certain properties of the nets; this is the topic of the next section.

6. Validation of RBN specifications

Validation of user requirements, i.e. elimination of ambiguity, inconsistency and incompleteness is a difficult process. Requirements specifications are valid if they are *deterministic, complete, consistent and correct*, and all these features can be ensured for RBN specifications. First of all, RBN specifications can be considered deterministic when the behaviour of the system model is always predictable; this follows from the premise that information systems are considered to behave deterministically, i.e., to produce a given result under given conditions. Secondly, RBN specifications can be considered complete if there is always something happening, whenever a signal arrives. Thirdly, RBN specifications can be considered consistent when there are no transitions with the same triggering signal and the same preconditions but with contradictory actions. Finally, RBN specifications are considered correct when they describe the behaviour that end-users expect of their system.

One way of validating ORM requirements specifications is by executing the corresponding C++ prototype whose generation was discussed in section 4. ORM requirements specifications can also be validated through analytical study and animation of the constructed RBN model. The executability of the RBN formalism and its potential for graphical animation provide invaluable help during the validation process. Animation has proven to be a valuable validation tool, and the results of using early animation show that it is a very promising means for reducing errors during requirements modelling and for ensuring that the intended system behaviour has been properly captured and modelled. Although animation provides remarkable assistance in the case of small systems, larger applications necessitate tools which can automatically detect contradictions, inconsistencies and redundancies in the model. The formal foundation of RBNs offers a sound basis for the development of such tools. A number of algorithms have been developed for proving various properties of a PrT net, like safeness, reachability, etc. [Murata 89], and since the RBN model is a variant of PrT nets and it can be automatically mapped to a specialised PrT net [Tsalgatidou 93] certain properties of the modelled system can be checked by exploiting PrT-net verification algorithms. For RBNs, the problem of proving that the system model can perform certain operations is reduced to the problem of proving that certain net markings are reachable from a given initial marking.

The above analysis motivates our current effort on developing the VENUS integrated environment for specification of object-oriented rule-based requirements, their mapping to RBN specifications and

animation and formal validation of RBNs. VENUS is being developed on Sun SPARC workstations running BSD Unix, X11R5 and OSF/Motif, in ANSI C++ and Prolog. The RBN graphical editor/animator uses graphical animation to validate RBN models in terms of determinism, completeness, consistency and correctness. A systems analyst dealing with small-scale applications may use this tool to TrunY realistic scenarios and Twhat-ifY cases for validation purposes. Missing and dead transitions and circular structures are easily detected, and this feedback is used for interactively redesigning the specifications. RBN specifications for large systems can be automatically validated by applying formal validation algorithms, currently under development.

7. Conclusions

The VENUS approach combines both the object-oriented and rule-based paradigms within a uniform framework, integrating all aspects of information systems. Two important advantages are that the final requirements specifications still represent explicitly application domain policy in terms of dynamic rules, and that the mapping of ORM schemas to C++ executable specifications and to the RBN model for validation purposes is formally defined and automated.

RBNs are validated through graphical animation, offering the advantages of rapid prototyping. In this way, requirements analysis can come up with easily modifiable, modular, clear, concise, compact and executable requirements specifications at the very early stages of system development. The sound foundation of RBNs makes this formalism a very good candidate notation for requirements specifications by allowing the use of formal validation tools and, subsequently, the mapping of requirements specifications to design and implementation structures (currently under development).

Two other interesting features of the RBN model are the ability to represent the dynamic behaviour of a system in different levels of abstraction and formality and the modelling of the behaviour of one object class in a single transition of a Thigh-level RBNY. This latter feature is compliant to the so-called Tlocalisation principleY suggesting to describe each object in isolation from the others [Rolland 92], since all the rules and signals related to one object are localised in a corresponding Thigh-level RBNY transition instead of being spread out in a huge RBN for the entire system, as is the case in classical behaviour models.

Although work on VENUS is still in progress, the current version of the system already demonstrates a number of capabilities that can be used to bridge the

gap between informal and formal specifications. The analyst is responsible for communicating with the end-user and entering the required information in the system. VENUS supports the analyst by offering tools (graphical and syntax-directed editors, C++ generator, RBN graphical editor/ animator) to construct and validate requirements specifications, and provides specific support for evolutionary construction of requirements, using the facilities of the underlying system to ensure that each change is carried out in a consistent way throughout the requirements specification process.

The viability of this approach is currently under testing by using it into two real world applications: the modelling of office-related activities in a public organisation and in a university department. The early construction of a TworkingY user requirements model seems to facilitate a lot the process of its validation and the results are very promising.

We believe that the work described in this paper contributes to the development of systems that satisfy the initial user requirements and that are flexible enough to incorporate and reflect any future changes. Future work will focus on the construction of a full-scale prototype that can be tested on real-world applications and on the development of algorithms for mapping RBN specifications to design and implementation structures.

Acknowledgements

The authors would like to thank Phoebus Vilanakis and Thanos Avdis for their contribution in the VENUS development.

References

- [Boehm 75] B.W. Boehm et al, Some Experiences with Automated Aids to the Design of Large-Scale Reliable Software, *IEEE Transactions on Software Engineering*, vol.1, no.2, February 1975.
- [Booch 92] G. Booch, *Object-Oriented Design and Applications*, 2nd edition, Addison-Wesley, 1992.
- [Brinkkemper 90] S. Brinkkemper & A.H.M. ter Hofstede, The Conceptual Task Model: a Specification Technique between Requirements Engineering and Program Development, *Proceedings of CAiSE '90*, LNCS, vol.436, Springer-Verlag, 1990, pp.228-250.
- [Coad 91] P. Coad & E. Yourdon, *Object-Oriented Analysis*, Prentice-Hall Int., 1991.
- [Coleman 92] D. Coleman, F. Hayes & S. Bear, Introducing Objectcharts or How to Use Objectcharts in Object-Oriented Design, *IEEE Transactions on Software Engineering*, vol.18, no.1, January 1992, pp.9-18.
- [Dahler 87] J. Dahler et al, A Graphical Tool for the Design and Prototyping of Distributed Systems, *ACM SIGSOFT*, vol.12, no.3, 1987, pp. 25-36.
- [D'Haenens 90] I. D'Haenens, F. van Assche, E. Halpin & B. Karakostas, Experiences with Rule-Based Dynamic Modelling, in [Sol 91], pp.231-239.
- [Durr 92] E. Durr & J. van Katwijk, VDM++ A Formal Specification Language for Object-Oriented Designs, *Proceedings of the 7th International Conference TOOLS EUROPE T92*, Dortmund, Germany, 1992, Prentice-Hall, pp.63-77.
- [Endres 75] A. Endres, An Analysis of Errors and their Causes in System Programs, *IEEE Transactions on Software Engineering*, vol.1, no.6, June 1975, pp.140-149.
- [Fickas 93] S. Fickas & A. Finkelstein (eds.), *Proceedings of IEEE International Symposium on Requirements Engineering*, San Diego, California, 4-6 Jan. 1993, IEEE Computer Soc. Press.
- [Finkelstein 92] A. Finkelstein & J. Kramer, TARA: Tool-Assisted Requirements Analysis, in [Loucopoulos 92], pp. 413-432.
- [Genrich 81] H.J. Genrich & K. Lautenbach, System Modelling with High-Level Petri Nets, *Theoretical Computer Science*, vol. 13, 1981, pp. 109-136.
- [Goguen 82] J. Goguen & J. Meseguer, Rapid Prototyping in the OBJ Executable Specification Language, *ACM SIGSOFT Software Engineering Notes*, vol.7, no.5, 1982, p.75.
- [Gouscos 92] D. Gouscos & A. Tsalgatidou, The ERM, ORM and RBN Models, Technical Report, Dept. of Informatics, University of Athens, May 1992.
- [Jones 86] C.B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall, 1986.
- [Kappel 91] G. Kappel & M. Schrefl, Using an Object-Oriented Diagram Technique for the Design of Information Systems, in [Sol 91], pp.97-129.
- [Kemmener 85] R.A. Kemmener, Testing Formal Specifications to Detect Design Errors, *IEEE Transactions on Software Engineering*, vol.11, no.1, January 1985, pp.32-43.
- [Lakos 93] C.A. Lakos & C.D. Keen, Modelling a Door Controller Protocol in LOOPN, *Proceedings of the 10th International Conference TOOLS EUROPE T93*, Versailles, France, 1993, Prentice-Hall, pp.31-44.
- [Loucopoulos 91] P. Loucopoulos, P. Mc Brien, F. Schumacker, C. Theodoulidis, V. Kopanas & B. Wangler, Integrating Database Technology, Rule-Based Systems and Temporal Reasoning for Effective Information Systems: The TEMPORA Paradigm, *Information Systems*, vol.1, no.1, April 1991.
- [Loucopoulos 92] P. Loucopoulos & R. Zicari (eds.), *Conceptual Modeling, Databases and CASE*, John Wiley & Sons, 1992.

- [Murata 89] T. Murata, Petri Nets: Properties, Analysis and Applications, *Proceedings of the IEEE*, vol.77, no.4, April 1989.
- [Obal 93] W.D. Obal II & W.H. Sanders, Importance Sampling Simulation in UltraSAN, *Simulation*, vol.61, no.5, November 1993.
- [Pezz• 92] M. Pezz• & C. Ghezzi, Cabernet: A Customizable Environment for the Specification and Analysis of Real-Time Systems, Dipartimento di Elettronica e dell' Informazione, Politecnico di Milano, September 1992.
- [Plat 92] N. Plat, J. van Katwijk & H. Toetnel, Application and Benefits of Formal Methods in Software Development, *Software Engineering Journal*, vol.7, no.5, September 1992, pp.335-346.
- [POSES 92] *Das Programmsystem POSES*, Gesellschaft fYr Proze\$automation & Consulting mbH Chemnitz, August 1992.
- [PROD 93], *PROD User's Guide*, Helsinki University of Technology, Digital Systems Laboratory, January 1993.
- [Reubenstein 91] H.B. Reubenstein & R.C. Waters, The Requirements Apprentice: Automated Assistance for Requirements Acquisition, *IEEE Transactions on Software Engineering*, vol., no.3, March 1991, pp. 226-240.
- [Rolland 92] C. Rolland, Trends and Perspectives in Conceptual Modelling, in [Loucopoulos 92], pp. 27-48.
- [Rumbaugh 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy & W. Lorensen, *Object-Oriented Modelling and Design*, Prentice-Hall, 1991.
- [Shand 88] J. Shand et al, An Enviroment for the Execution and Graphical Animation of JSD Specifications, in *Procs. of International Workshop of KBS in S/W Engineering*, UMIST, Manchester, 1988.
- [Sol 91] H.G. Sol & K.M. van Hee (eds.), *Dynamic Modelling*, North-Holland, 1991.
- [Spivey 87] M. Spivey, *The Z Notation: A Reference Manual*, Oxford University Computing Laboratory Programming Research Group, 1987.
- [Tsalgatidou 91] A. Tsalgatidou & P. Loucopoulos, An Object-Oriented Rule-Based Approach to the Dynamic Modelling of Information Systems, in [Sol 91], pp.131-147.
- [Tsalgatidou 93] A. Tsalgatidou, D. Gouscos & C. Halatsis, Rule-Based Behaviour Modelling of Information Systems, *Proceedings of the 26th Hawaiian Conference on Systems Sciences (HICSS-26)*, vol.IV, January 1993, IEEE Computer Society Press, pp. 409-418.
- [van Baelen 92] S. van Baelen, J. Lewi, E. Steegmans & H. van Riel, EROOS: An Entity-Relationship Based OO Specification Method, *Proceedings. of the 7th International Conference TOOLS EUROPE T92*, Prentice-Hall, 1992, pp.103-118.
- [Zave 91] P. Zave, An Insider's Evaluation of PAISLey, *IEEE Transactions on Software Engineering*, vol.17, no.3, March 1991, pp. 212-225.

A. Appendix

```
/******\
* CUSTOMER ORDER: CLASS DEFINITION *
\*****/

class META_CUSTOMER_ORDER:
public META_ORDER
{
/* CLASS BEHAVIOUR */
public:
void CBU_ProcessOrdersOnHold (OBJECTID Product) ;
/* CLASS BEHAVIOUR RULES */
protected:
void CR_ProcessOrdersOnHold_R1
(OBJECTID Product) ;
/* CLASS PROCEDURES */
private:
void CP_NotifyOrders
(OBJECTID Product, PRIORITY Priority) ;
/* INSTANCE RETRIEVAL MECHANISM */
public:
OBJECTID Create (CUSTOMER_ORDER &Obj) ;
void Update (CUSTOMER_ORDER &Obj) ;
void Delete (OBJECTID ObjId) ;
void Retrieve
(OBJECTID ObjId, CUSTOMER_ORDER &Obj) ;
void RetrieveFirst (CUSTOMER_ORDER &Obj) ;
void RetrieveNext (CUSTOMER_ORDER &Obj) ;
```

```
};

class CUSTOMER_ORDER: public ORDER
{
/* CUSTOMER_ORDER: CLASS-PART DEFINITION */
public:
static META_CUSTOMER_ORDER C;
/* INSTANCE PROPERTIES */
TIME ArrivalTime;
PRIORITY Priority;
OBJECTID Customer;
OBJECTID Invoice;
/* INSTANCE BEHAVIOUR */
void IBU_ProcessNow (void);
/* INSTANCE BEHAVIOUR RULES */
protected:
void IR_ProcessNow_R1 (void);
void IR_ProcessNow_R2 (void);
/* INSTANCE PROCEDURES */
private:
void IP_Set_Priority (PRIORITY Priority);
};

/*****\
* CUSTOMER ORDER: CLASS-PART METHODS *
\*****/

void META_CUSTOMER_ORDER::
    CBU_ProcessOrdersOnHold (OBJECTID Product)
{
    CR_ProcessOrdersOnHold_R1(Product);
}

void META_CUSTOMER_ORDER::
    CR_ProcessOrdersOnHold_R1 (OBJECTID Product)
{
    CP_NotifyOrders(Product, OnHold);
}

void META_CUSTOMER_ORDER::CP_NotifyOrders (OBJECTID Product, PRIORITY Priority)
{
    CUSTOMER_ORDER CustOrder;
    PRODUCT Prod;

    CUSTOMER_ORDER::C.RetrieveFirst(CustOrder);
    while (CustOrder.ObjectId)
    {
        if ((CustOrder.Product == Product) && (CustOrder.Priortity == Priority))
        {
            PRODUCT::C.Retrieve(Product,Prod);
            Prod.IP_SetPriority(Immediate);
            CustOrder.IBU_ProcessNow();
        }
        CUSTOMER_ORDER::C.RetrieveNext(CustOrder);
    }
}

/*****\
* CUSTOMER ORDER: INSTANCE-PART METHODS *
\*****/

void CUSTOMER_ORDER::IBU_ProcessNow (void)
{
    IR_ProcessNow_R1 ();
    IR_ProcessNow_R2 ();
}
```

```
}

void CUSTOMER_ORDER::IR_ProcessNow_R1 (void)
{
PRODUCT Prod ;

PRODUCT::C.Retrieve (this -> Product, Prod) ;
if (this -> Quantity > Prod.StockQty)
{
IP_SetPriority(OnHold) ;
EXT_ENV_NoShipment(this -> OrderNo) ;
}
}

void CUSTOMER_ORDER::IR_ProcessNow_R2 (void)
{
PRODUCT Prod ;

PRODUCT::C.Retrieve (this -> Product, Prod) ;
if (this -> Quantity <= Prod.StockQty)
{
IP_SetPriority(Processed) ;
INVOICE::C.IssueInvoice(this -> ObjectId) ;
Prod.DecreaseStock(this -> Quantity) ;
}
}

void CUSTOMER_ORDER::IP_SetPriority
(PRIORITY Priority)
{
this -> Priority = Priority ;
C.Update(*this) ;
}
}
```