HKU CS Tech Report TR-92-05

The 5th International Congress on Computational and Applied Mathematics (ICCAM '92), Leuven, Belgium (1992)

Towards a 3-dimensional Net-based Object-Oriented DeveLopment Environment (NOODLE) [1]

T.H. Tse [2] and C.P. Cheng
Department of Computer Science
The University of Hong Kong
Pokfulam Road
Hong Kong

(Email addresses: thtse@cs.hku.hk)

ABSTRACT

Object-oriented analysis and design methodologies are considered as the most popular software development methods for the 1990s. Numerous graphic notations have already been designed for this purpose. A common drawback, however, is that they have been developed informally. There is no theoretical framework enabling us to define precisely the object-oriented concepts involved, to solve concurrency problems and to verify the correctness of the implementation. Although a number of formal object-oriented specification languages have been proposed by academics, they are not linked with the popular methodologies. Practitioners are reluctant to use these unfamiliar formal tools.

We propose a 3-dimensional net structure behind object-oriented analysis and design. The concepts of classes, objects, inheritance, overloading and message passing can be modelled within the framework. Some of the concepts can be visualized as a projection of the general model into 2-dimensional space. The model can be implemented in terms of existing object-oriented graphic notations. A development environment using these notations as front-end user-interface can be developed so that the formal framework is transparent to users.

---

1. This project is supported in part by a grant of the Research Grants Council and a CRCG grant of the University of Hong Kong.

2. On leave at the Programming Research Group, University of Oxford.

1. INTRODUCTION

A good development methodology is the key towards successful computer systems. It should be user-friendly so that users may feel comfortable in using it and, at the same time, vigorous in its definition so that validation and verification of systems can be carried out easily [13, 19].

Object-oriented analysis and design have emerged quickly with quite a number of graphic notations and methodologies being proposed [5, 7, 8, 18, 20, 4]. User-friendliness is the main target and focal point. One common drawback is that they have been developed informally. Systems cannot be defined precisely and systems implementation cannot be validated and verified without a formal background. On the other hand, a number of formal object-oriented specification languages have been developed, independently of software development methodologies [1, 11, 17]. Practitioners are, however, rather reluctant to use these formal tools since unfamiliar languages are involved [3].

To bridge the gap between object-oriented graphic notations and formal languages, a 3-dimensional Net-based Object-Oriented DeveLopment Environment (NOODLE) is proposed. One of the advantages of net theory is that it has both graphic and algebraic representations. Hence, models defined by net theory can cater for both visualization and formalism. General object-oriented concepts including classes, objects, inheritance, overloading and message passing are modelled so that any object-oriented notations can be mapped into this formal framework.

In Section 2, we will briefly describe object-oriented features. The justification of using net theory as our modelling language will be described in Section 3, followed by an outline of net theory, the framework behind the model. In Section 5, we will describe with examples details of the NOODLE model. Both the graphic and algebraic representations of the model will be given.

2. OBJECT-ORIENTEDNESS

We would like to introduce the important concepts which are generally accepted as the features of object-oriented development.

(i) Classes

A class refers to a group of objects which share similar properties. Such properties include a set of attributes and a set of methods. Attributes represent the states of the object and methods are operations or services provided by the class. Hence, methods are the only interfaces through which the corresponding objects can be accessed while values of attributes can only be accessed and transformed by methods of the class. For example, we can have a class *car* which has attributes like *speed* and *cylinderCapacity*, and methods like *start*, *accelerate* and *brake*.

(ii) Objects

An object is an instance of a class. Objects of the same class differ from one another by their own identities and states. Objects like *FordEscort*, *HondaCivic* and *AustinRover* are all instances of the class *car*.

(iii) Inheritance

Inheritance refers to the relationship between two classes whereby one specific class acquires the structure of another. Classes which acquire the structures of others are called subclasses and classes whose structures are being acquired are called superclasses. For example, we can have a

class *fourWheelDriveCar* as a subclass of the class *car* since a *fourWheelDriveCar* is a specialization of a *car*.

(iv) Overloading

It refers to the ability to use the same name to refer to different methods defined in classes which have a superclass-subclass relationship. Thus, a method likes *accelerate* in the class *fourWheelDriveCar* can have the same name as that in the class *car* but different meaning since four wheels are involved instead of two.

(v) Message Passing

Communication between objects is achieved through message passing. A message is a request from object *A* to object *B* to perform one of *B*'s operations. *A* and *B* may also be called an actor and a server respectively. Details of operations performed by *B* are not known to *A*. For example, when accelerating, a message *increaseRevolutionRate*, which is responsible for increasing the rate of revolution of the wheels, may have to be passed from an object of the class *car* to another object of the class *accelerator*. Messages can also be passed within an object such that requests of operations are sent from and received by methods belonging to the same object.

3. JUSTIFICATIONS OF USING NET THEORY

A model captures important characteristics of a real world system which can then be understood through the manipulation and study of various components, features and activities of the relatively simpler model. Thus, important characteristics, which at least include the structural and dynamic properties, of systems should be modelled as closely and as precisely as possible. In NOODLE, we have chosen predicate/transition nets, a type of high level Petri nets, to model features in object-oriented system development. Net theory based on such nets is a powerful formal modelling language. It is especially good in modelling systems featuring parallelism, concurrency and dynamic properties. The following are the reasons for choosing net theory as our modelling language.

(i) Theoretically speaking, an object-oriented system consists of a number of distinct objects, which run independently at the same time unless interacted on by other objects. Net theory is especially good in modelling systems which involve distributed components.

(ii) Nets have a formal syntax and semantics by which both the structural and dynamic properties of systems can be modelled. At the same time, nets have a graphic representation which preserves the pictorial aspect of popular object-oriented development notations and makes the language more user-friendly.

(iii) One deficiency of structured analysis and design techniques [9, 14] is that different graphic notations are required in different system development stages. For instance, a number of tools such as data flow diagrams, data dictionaries and mini-specifications are being used in the analysis phase, and structured charts are used in the design phase. As a result, a tool need to be converted into another during systems development. Consistency problems often arise. This deficiency is mainly due to the fact that one single tool or notation is not powerful enough to describe both structural and dynamic properties of systems. Since nets can model both structural and dynamic properties, there is no need for a second modelling tool.

(iv) Object-oriented development emphasizes both data and operations. But most of the graphic tools can only model one aspect. For example, data flow diagrams show the transitions between processes, and state-transition diagrams show the transitions between states of data. Nets, on the other hand, model data with circles and processes with bars on the same level of importance.

(v) Unlike structured development methods, which emphasize only top-down design, object-oriented development favour also bottom-up techniques. In net theory, the concepts of net refinement and abstraction can support both top-down and bottom-up techniques.

(vi) Class reuse is one of the features of object-oriented development. Embedding of nets can support this concept by connecting reusable parts to newly defined parts.

(vii) The instantiation of objects requires a specific class definition to be extracted from the class hierarchy. This can be supported by the concept of sectioning in net theory.

A few other projects [12, 2, 6] similarly adopt a net approach to model object-oriented systems. Unlike our comprehensive approach, however, these project either do not model inheritance, or do so only at a syntactic level.

4. NET THEORY

4.1 Basic Definition of Petri Nets

The definition of a Petri net [15, 16] can be divided into two parts, structural and dynamic:

(a) The structure of a Petri net is composed of three types of components: a set of places P, a set of transitions E and a set of flow relations F between elements of the sets P and E. Thus, $C = (P, E; F)$ is a Petri net if and only if

   (i) $P \cap E = \emptyset$ and $P \cup E \neq \emptyset$

   (ii) $F \subseteq (P \times E) \cup (E \times P)$

   Graphically, we represent places by circles, transitions by bars, and flow relations by arcs between circles and bars. Figure 1(a) shows an example of a simple Petri net.

(b) Dynamically, tokens can be put into places to denote that the corresponding condition has been satisfied. Firing of transitions move tokens from places to places. For $e \in E$, let

   (i) $\bullet e$ denote the pre-condition $\{a \in P \mid \langle a, e \rangle \in F\}$

   (ii) $e\bullet$ denote the post-condition $\{a \in P \mid \langle e, a \rangle \in F\}$

   (iii) c be a subset of P, called a case

   Then $e \in E$ is said to be c-enabled if and only if $\bullet e \subseteq c$ and $e\bullet \cap c = \emptyset$. e can be fired if it is c-enabled. $c' = (c \setminus \bullet e) \cup e\bullet$ is called the follower case of c under e. In Figure 2(a), for example, t_1 is $\{p_1, p_3\}$-enabled and hence fired. $\{p_2, p_4\}$ is the follower case, as shown in Figure 1(b).

Refinement of places and transitions is also allowed. A net $C' = (P', E'; F')$ is a refinement of C if and only if

(i) $P \subseteq P'$

(ii) $E \subseteq E'$

(iii) $F = F' \cap ((P \cup E) \times (P \cup E))$

(iv) There exists an abstraction function g: $(P' \cup E') \rightarrow (P \cup E)$ which maps places or transitions of the refinement C' to places or transitions of the net C.

4.2 Definition of Predicate/Transition Nets

Predicate/transition nets [16, 10] are high-level Petri nets in which the movement of tokens is replaced by the valuation of predicates. In order to define this kind of nets, we need the concepts of algebras and terms.

(i) Let D be an arbitrary set and let Φ be a set of partial operations σ: $D^n \rightarrow D$. Then $\boldsymbol{D} = (D, \Phi)$ is called an algebra. In particular, Φ may contain constant operations d: $D^0 \rightarrow D$, which may be identified with the elements of D.

(ii) Let X be a set of variables. The set $\Sigma_{\boldsymbol{D}}(\mathrm{X})$ of terms of D over X is the smallest set of expressions such that

(a) $X \subseteq \Sigma_{\boldsymbol{D}}(\mathrm{X})$

(b) For any terms $t_1, ..., t_n \in \Sigma_{\boldsymbol{D}}(\mathrm{X})$ and for any operation σ: $D^n \rightarrow D \in \Phi$, the term $\sigma(t_1, ..., t_n) \in \Sigma_{\boldsymbol{D}}(\mathrm{X})$. In particular, an element $d \in D$, which can be considered as a constant operation d: $D^0 \rightarrow D$, is a term.

(iii) A mapping β: $X \rightarrow D$ is called a valuation of X. It induces, canonically, a mapping β: $\Sigma_{\boldsymbol{D}}(\mathrm{X}) \rightarrow D$ by $\beta(\sigma(t_1, ..., t_n)) = \sigma(\beta(t_1), ..., \beta(t_n))$.

Using these notions, we are now able to define the structure of predicate/transition nets. $C = (P, E; F, \boldsymbol{D}, \lambda, c)$ is called a predicate/transition net if and only if

(i) $(P, E; F)$ is a net. The elements of P and E are called predicates and events, respectively.

(ii) $\boldsymbol{D}$ is an algebra.

(iii) λ: $F \rightarrow 2^{\Sigma_{\boldsymbol{D}}(\mathrm{X})} \setminus \{\varnothing\}$ is a mapping.

(iv) There exists a mapping c: $P \rightarrow 2^D$, known as the initial case of C.

The dynamic part of predicate/transition nets is defined as follows:

(i) Let $e \in E$ and let β be a valuation such that, for all $f \in F \cap ((P \times \{e\}) \cup (\{e\} \times P))$, if $t_1, t_2 \in \lambda(f)$ and $t_1 \neq t_2$, then $\beta(t_1) \neq \beta(t_2)$. For a given mapping c: $P \rightarrow 2^D$, known as a case, e is called c-enabled with β if and only if

(a) $\beta(\lambda(<p, e>)) \subseteq c(p)$ for all $p \in \bullet e$

(b) $\beta(\lambda(<e, p>)) \cap c(p) = \varnothing$ for all $p \in e\bullet$

(ii) An event e which is c-enabled with β yields a follower case c' of c under β by

$$c'(p) = \begin{cases} c(p) \setminus \beta(\lambda(< p, e >)) & \text{iff } p \in \bullet e \setminus e\bullet \\ c(p) \cup \beta(\lambda(< e, p >)) & \text{iff } p \in e\bullet \setminus \bullet e \\ c(p) \setminus \beta(\lambda(< p, e >)) \cup \beta(\lambda(< e, p >)) & \text{iff } p \in \bullet e \cap e\bullet \\ c(p) & \text{otherwise} \end{cases}$$

5. THE NOODLE MODEL

In this section, we are going to present the way how each of the object-oriented ingredients is modelled.

5.1 Classes

We can define the concept of classes using algebras and terms. Let A be a set of attributes, B be a set of messages and E be a set of methods. $C = (P, E; F, \boldsymbol{D}, \lambda)$ is a class if and only if

(i) $(P, E; F)$ is a net such that

- (a) $P = A \cup B$
- (b) $F = G \cup H$, where $G \subseteq (A \times E) \cup (E \times A)$ is a binary relation showing the information flows between attributes and methods of the class, and $H \subseteq (B \times E) \cup (E \times B)$ is a binary relationship showing the messages passing to and from methods.

(ii) $\boldsymbol{D} = (D, \Phi)$ is an algebra, where D is the set of possible message parameter values and Φ is the set of partial operations on D.

(iii) λ: $F \rightarrow 2^{\Sigma_{\boldsymbol{D}}(X)} \setminus \{\varnothing\}$, where X is a set of variables to which message parameters are applied.

Methods can be refined using the concept of net refinement. Thus, $e = (MP, ME; MF, \boldsymbol{MD}, \mu) \in E$ is the refinement of a method if and only if the following conditions are satisfied:

(i) $(MP, ME; MF)$ is a net such that

- (a) MP is the set of local states and ME is the set of local processes of the method.
- (b) $MF \subseteq (MP \times ME) \cup (ME \times MP)$ is a binary relation which shows the information flows between local states and processes within the method.

(ii) $\boldsymbol{MD} = (MD, \Gamma)$ is an algebra, where MD is the set of possible message parameters and states for the method and Γ is the set of partial operations on MD.

(iii) μ: $MF \rightarrow 2^{\Sigma_{MD}(MX)} \setminus \{\varnothing\}$, where MX is a set of variables to which actual parameters and states are applied.

According to this concept of method refinement, a refined class can now be defined. Let $C = (P, E; F, \boldsymbol{D}, \lambda)$ be a class, and let $e_i = (MP_i, ME_i; MF_i, \boldsymbol{MD_i}, \mu_i) \in E$ for $i = 1, \ldots n$, where n is the number of methods in C. Then $C' = (P', E'; F', \boldsymbol{D}', \lambda')$ is the refined class of C if and only if

(i) $P' = P \cup MP_1 \cup ... \cup MP_n$

(ii) $E' = ME_1 \cup ... \cup ME_n$

(iii) $F' = F \cup MF_1 \cup ... \cup MF_n$

(iv) $\mathbf{D}' = ((D \cup MD_1 \cup ... \cup MD_n), (\Phi \cup \Gamma_1 \cup ... \cup \Gamma_n))$, where MD_1, ..., MD_n are the sets of possible message parameters and states for the corresponding methods, and each Γ_i is the set of partial operation on MD_i.

(v) λ': $F' \rightarrow 2^{\Sigma_{\mathbf{D}'}(X')} \setminus \{\varnothing\}$, where $X' = X \cup MX_1 \cup ... \cup MX_n$ and the MX_i's are set variables.

Moreover, the refined class C' can be mapped to the class C under the abstraction function g: $(P' \cup E') \rightarrow (P \cup E)$.

For example, Figure 2 shows the graphic representation of the class *list*. The components of the class are encapsulated by a box within which methods and attributes are defined. Methods are represented by transitions and attributes are represented by predicates. Flow relationships between transitions and predicates represent the flow of messages and parameters. We can see that three methods, namely *initialize, get* and *put*, and one attribute, namely *items*, are defined for *list*. Moreover, messages, together with parameters, are defined through predicates. Thus, each of the methods *initialize* and *put* have one input message predicate, namely *initializeList* and *putList* respectively, through which messages and parameters are sent to the class. In addition, an output message predicate *returnedItem* is defined for the method *get* to return results to the message originator.

This graphic representation can be transformed into the following algebraic representation: *list* = $(P, E; F, \mathbf{D}, \lambda)$, where

(i) $P = A \cup B$ for A = {*items*} and
B = {*initializeList*, *putList*, *getList*, *returnedItem*}

(ii) E = {*initialize*, *put*, *get*}

(iii) F = {<*initializeList*, *initialize*>, <*initialize*, *items*>, <*putList*, *put*>, <*itmes*, *put*>, <*put*, *items*>, <*getList*, *get*>, <*items*, *get*>, <*get*, *items*>, <*get*, *returnedItem*s>}

(iv) $\mathbf{D} = (D, \Phi)$ is an algebra, where $D = A \cup B \cup N$ for A = the set of possible list items, B = the set of possible list values, N = the set of natural numbers and Φ = {*put*, *get*, *item*, Λ}.

(v) λ is a mapping which maps each arc to one or more operations such that

λ(<*initializeList*, *initialize*>) = {*empty*}
λ(<*initialize*, *items*>) = {*l*}
λ(<*putList*, *put*>) = {*i*, *p*}
λ(<*items*, *put*>) = {*l*}
λ(<*put*, *items*>) = {*put*(*i*, *l*, *p*)}
λ(<*getList*, *get*>) = {*p*}
λ(<*items*, *get*>) = {*l*}
λ(<*get*, *items*>) = {*get*(*l*, *p*)}
λ(<*get*, *returnedItem*>) = {*item*(*l*, *p*)}

After defining the class *list* using both graphic and algebraic representations of NOODLE, methods in the class can now be defined through net refinement. Figure 3 shows the graphic representation of the refined method *put* of the class *list*. Like a class, a method is also encapsulated by a box within which local processes are represented by transitions and states by predicates. Within the method *put*, we define three local processes *readItem*, *readPosition* and *putItem*, and two local states *position* and *item*. The methods *initialize* and *get* can be refined in a similar way.

Apart from a graphic representation, methods can also be defined in an algebraic form. For example, $put \in E$ is refined into $(MP_{put}, ME_{put}; MF_{put}, \boldsymbol{MD_{put}}, \mu_{put})$, where

(i) $MP_{put} = \{item, position\}$

(ii) $ME_{put} = \{readItem, readPosition, putItem\}$

(iii) MF_{put} = {<*readItem*, *item*>, <*readPosition*, *position*>,
<*item*, *putItem*>, <*position*, *putItem*>}

(iv) $\boldsymbol{MD_{put}} = (MD_{put}, \Gamma_{put})$ is an algebra such that $MD_{put} = A \cup N$ for A = the set of list items, N = the set of natural numbers and $\Gamma_{put} = \varnothing$

(v) μ_{put} is a mapping which maps each arc to one or more operations such that

μ_{put}(<*getItem*, *item*>) = $\{i\}$
μ_{put}(<*getItem*, *position*>) = $\{p\}$
μ_{put}(<*item*, *putItem*>) = $\{i\}$
μ_{put}(<*position*, *putItem*>) = $\{p\}$

After all the methods of the class *list* have been refined, a refined class *list′* can be defined by combining the original *list* definitions with the definitions of the refined methods. The algebraic representation of the refined class *list′* is $(P', E'; F', \boldsymbol{D'}, \lambda')$, where

(i) $P' = P \cup MP_{put} \cup MP_{get} \cup MP_{init}$

(ii) $E' = ME_{put} \cup ME_{get} \cup ME_{init}$

(iii) $F' = F \cup F_{put} \cup F_{get} \cup F_{init}$

(iv) $\boldsymbol{D'} = ((D \cup MD_{put} \cup MD_{get} \cup MD_{init}), (\Phi \cup \Gamma_{put} \cup \Gamma_{get} \cup \Gamma_{init}))$

(v) $\lambda' = F' \rightarrow 2^{\Sigma_{\boldsymbol{D'}}(X')} \setminus \{\varnothing\}$, where $X' = X \cup MX_{put} \cup MX_{get} \cup MX_{init}$

Moreover, the refined class *list′* can be mapped to the class *list* under the abstraction function $g: P' \cup E' \rightarrow P \cup E$ such that $g(x) = put$, $g(y) = get$ and $g(z) = initialize$, where $x \in MP_{put} \cup ME_{put}$, $y \in MP_{get} \cup ME_{get}$ and $z \in MP_{init} \cup ME_{init}$.

5.2 Objects

Objects are instances of classes. They are the running entities of their corresponding class definitions. Their structures are the same as their corresponding classes with the difference that initial cases are provided for objects. Each initial case for a class is the state with which the object starts in the first place. Thus, objects can be defined as initial cases in addition to their corresponding class definitions. $O = (C, c)$ is an object if and only if

(i) C is a class

(ii) $c: P \to 2^D$ is an initial case of the object O, where D is the underlying set for the algebra $\boldsymbol{D}$ in the class C.

For example, an object l, which is an instance of the class *list*, can be instantiated from the class *list* by first making a copy of the class *list* and then providing an initial case for the particular instance. Algebraically speaking, the object l can be initially defined as (*list*, c), where

(i) $list = (P, E; F, \boldsymbol{D}, \lambda)$

(ii) $c = \Lambda$.

5.3 Inheritance and Overloading

Inheritance is a relationship between two classes C and CC such that CC acquires the structure of C. Thus, class CC makes use of the methods and attributes of class C and provides services inherited from C. C is known as the superclass of CC and CC is called the subclass of C. On the other hand, inheritance supports overloading in the sense that some methods and attributes of class C can be redefined in class CC and thus provide services which have the same name but perform differently.

Let us first define the relationship between superclasses and subclasses. Let $C = (P, E; F, \boldsymbol{D}, \lambda)$ and $CC = (PP, EE; FF, \boldsymbol{DD}, \lambda\lambda)$ be two different classes such that

(i) $P = A \cup B$, $PP = AA \cup BB$

(ii) $F = G \cup H$, $FF = GG \cup HH$

(iii) $\boldsymbol{D} = (D, \Phi)$, $\boldsymbol{DD} = (DD, \Phi\Phi)$

(iv) $\lambda: F \to 2^{\Sigma_{\boldsymbol{D}}(X)} \setminus \{\varnothing\}$, $\lambda\lambda: FF \to 2^{\Sigma_{\boldsymbol{DD}}(XX)} \setminus \{\varnothing\}$

If CC is inherited from C, i.e., if CC is a subclass of C, then

(i) $A \subseteq AA$

(ii) $E \setminus \{e_i \in E \mid e_i \text{ is an overloaded method}\} \subseteq EE$,

(iii) $G \subseteq GG$

(iv) $D \subseteq DD$ and $\Phi \subseteq \Phi\Phi$

(v) $X \subseteq XX$

Further conditions are required for the inheritance and overloading of methods. Suppose $e_{CC} \in E$ is a method in class CC inherited from $e \in E$ in class C.

(i) If e only accepts input messages through the predicate m^i without returning messages to the originators, we add the following predicates and flow relations to subclass CC (see Figure 4):

$$m^i_{CC} \in PP \text{ and } \langle m^i_{CC}, e_{CC}\rangle \in FF$$
$$e^i \in PP \text{ and } \langle e_{CC}, e^i\rangle \text{ and } \langle e^i, e\rangle \in FF$$

(ii) If e accepts input messages through the predicate m^i and returns messages to the originators through the predicate m^o, we add the following predicates and flow relations to subclass CC (see Figure 5):

$$m^i_{CC} \in PP \text{ and } <m^i_{CC}, e_{CC}> \in FF$$
$$e^i \in PP \text{ and } <e_{CC}, e^i> \text{ and } <e^i, e> \in FF$$
$$e^o \in PP \text{ and } <e, e^o> \text{ and } <e^o, e_{CC}> \in FF$$
$$m^o_{CC} \in PP \text{ and } <e_{CC}, m^o_{CC}> \in FF$$

and the refinement of e_{CC} is $(MP, ME; MF, \boldsymbol{MD}, \mu)$, where $MP = \varnothing$, $MF = \varnothing$ and $ME = \{e'_{CC}, e''_{CC}\}$ for e'_{CC} being a process accepting input messages and e''_{CC} being a process returning messages.

Now, let $e \in E$ be a method defined in class C and overloaded by a method $e_{CC} \in EE$ defined in class CC. Like defining a new method in any class, a transition e_{CC} and corresponding messages paths must be added in subclass CC. Moreover, instead of accessing attributes through methods of superclasses, like inherited methods, the method e_{CC} of the subclass accesses the corresponding attributes directly. That is, direct connections between the transition e_{CC} and the corresponding predicates are added. Like inherited methods, two categories of overloaded methods must be considered:

(i) If e only accepts input messages through the predicate m^i without returning messages to the originators, and if $a \in P$ is the attribute accessed by e, we add the following predicate, transition and flow relations to subclass CC (see Figure 6):

$$m^i_{CC} \in PP \text{ and } <m^i_{CC}, e_{CC}> \in FF$$
$$<e^{CC}, a> \in FF \text{ if } <e, a> \in F$$
$$<a, e^{CC}> \in FF \text{ if } <a, e> \in F$$

(ii) If e accepts input messages through the predicate m^i and returns messages to the originators through the predicate m^o, and if $a \in P$ is the attribute accessed by e, we add the following predicates, transition and flow relations to subclass CC (see Figure 7):

$$m^i_{CC} \in PP \text{ and } <m^i_{CC}, e_{CC}> \in FF$$
$$<e^{CC}, a> \in FF \text{ if } <e, a> \in F$$
$$<a, e^{CC}> \in FF \text{ if } <a, e> \in F$$
$$m^o_{CC} \in PP \text{ and } <e_{CC}, m^o_{CC}> \in FF$$

In addition, the set of functions for the algebra ***DD*** is transformed by augmenting the set of operations and possibly the set of possible message parameters for the algebra ***D*** with operations for the additional flow relations defined above and possible message parameters.

Let us look at an example. The class *list* is a general list in which items of any kind can be put into the list. On the other hand, the class *nameList* is a special case of *list* such that elements are restricted to names only. Figure 8 shows how *nameList* and *list* are represented graphically in a 3-dimensional fashion. The definition of the superclass *list* lies on the top platform while the subclass *nameList* lies on the bottom platform of the cube. Figure 9(a) illustrates the fact that the method *initialize* of the class *list* is inherited by the subclass *name list* since the initialization process for both classes are the same. A transition $initialize_{nameList}$ is thus added to the bottom platform, together with an input messages path towards it and a connection path towards the transition *initialize*. In Figures 9(b) and

9(c), the methods *put* and *get* of *list* are overloaded by *nameList* since *put* and *get* of *list* cater for all kinds of list elements while the *put* and *get* methods of the class *nameList* cater for records of names only. Transitions $put_{nameList}$ and $get_{nameList}$ are thus added to the bottom platform, together with input and output messages paths and access paths which directly link the transitions and the corresponding attributes. These constructions are in fact part of an overall NOODLE model of classes. They show the contents of classes as well as the class hierarchy. Visualization is made simple by showing only the components on selected 2-dimensional planes. This can be achieved using the proposed development environment.

The corresponding algebraic representation of the class *nameList* is (P, E; F, $\boldsymbol{D}$, λ), where

(i) $P = \{items, initializeList_{nameList}, putList_{nameList}, getList_{nameList}, returnedItem_{nameList}\}$

(ii) $E = \{initialize_{nameList}, put_{nameList}, get_{nameList}\}$

(iii) $F = \{<initializeList_{nameList}, initialize_{nameList}>, <putList_{nameList}, put_{nameList}>, <items, put_{nameList}>, <put_{nameList}, items>, <getList_{nameList}, get_{nameList}>, <items, get_{nameList}>, <get_{nameList}, getItem_{nameList}>, <get_{nameList}, items>\}$

(iv) $\boldsymbol{D} = (D, \Phi)$ is an algebra, where $D = A \cup B \cup N$ for A = the set of possible list items, B = the set of possible list values, N = the set of positive integers and $\Phi = \{put, get, item, \Lambda\}$

(v) λ is a function which maps each arc to one or more operations such that

$$\lambda(<initializeList_{nameList}, initialize_{nameList}>) = \{\Lambda\}$$
$$\lambda(<initialize_{nameList}, initialize^{i}>) = \{\Lambda\}$$
$$\lambda(<initialize^{i}, initialize>) = \{\Lambda\}$$
$$\lambda(<initialize, items>) = \{l\}$$
$$\lambda(<putList_{nameList}, put_{nameList}>) = \{i, p\}$$
$$\lambda(<items, put_{nameList}>) = \{l\}$$
$$\lambda(<put_{nameList}, items>) = \{put(i, l, p)\}$$
$$\lambda(<getList_{nameList}, get_{nameList}>) = \{p\}$$
$$\lambda(<items, get_{nameList}>) = \{l\}$$
$$\lambda(<get_{nameList}, getItem_{nameList}>) = \{item(l, p)\}$$
$$\lambda(<get_{nameList}, items>) = \{get(l, p)\}$$

5.4 Message Passing

Communication between objects is achieved through message passing. The NOODLE definition of message passing can be partitioned into two parts. The static part is the definition of all possible message paths. The dynamic part shows when and what messages are passed.

For the static part, each message path is modelled by connecting the corresponding transitions by a predicate. Messages can be classified into two categories:

(i) For Messages sent from one Object to Another

Let $O = (C, c)$ and $OO = (CC, cc)$ be two different objects where $C = (P, E; F, \boldsymbol{D}, \lambda)$ and $CC = (PP, EE; FF, \boldsymbol{DD}, \lambda\lambda)$ are classes. The possible messages paths between the two objects consist of the following:

(a) A set of predicates $B \cap BB$, where B is the set of messages in P and BB is the set of messages in PP

(b) A set of arcs $\{\langle \bullet x, x\rangle \mid x \in B \cap BB\} \cup \{\langle x, x\bullet\rangle \mid x \in B \cap BB\}$

(ii) For Messages within an Object

Let $e = (MP, ME; MF, \boldsymbol{MD}, \mu)$ and $ee = (MPP, MEE; MFF, \boldsymbol{MDD}, \mu\mu)$ be two methods in an object. The possible messages paths between them consist of the following:

(a) A set of predicates $MP \cap MPP$ which represents the possible messages

(b) A set of arcs $\{\langle \bullet x, x\rangle \mid x \in B \cap BB\} \cup \{\langle x, x\bullet\rangle \mid x \in B \cap BB\}$, where B is the set of messages for e and BB is the set of messages for ee

Dynamically, the passing of messages can simply be defined by the standard firing rules of predicate/transition nets. A message can only be passed to the destination if the method of the destination is enabled.

In Figure 10, for example, a teacher wants to get a student name from a name list. Thus two methods, *getName* and *get* are involved and they belong to the objects *teacher* and *nameList*, respectively. The object *teacher* collects a name by performing the method *getName* which requires a service that releases a name from the object *nameList*. Thus, the method *getName* of the object *teacher* initiates and sends a message *requestName* to the object *nameList* and requests for the service *get*, which releases a student name from the list of names and sends it back to the object *teacher* through the message predicate *returnName*.

Algebraically speaking, we can define the possible message paths between the two objects *teacher* $= (P, E; F, \boldsymbol{D}, \lambda, c)$ and *nameList* $= (PP, EE; FF, \boldsymbol{DD}, \lambda\lambda, cc)$, thus:

(i) The set of predicates $P \cap PP = \{requestName, returnName\}$

(ii) The set of arcs {<*getName*, *requestName*>, <*requestName*, *get*>, <*get*, *returnName*>, <*returnName*, *getName*>}

After defining the message paths between the two methods *getName* and *get*, we have to consider the conditions before and after sending the message. For example, in order to pass a message from *getName* to *get*, since the pre-condition of *get* is $\bullet get = \{requestName, item\}$, the following two conditions must be satisfied:

(i) $\beta(\lambda(\langle requestName, get\rangle)) \subseteq c(requestName)$

(ii) $\beta(\lambda(\langle items, get\rangle)) \subseteq c(items)$

After a message has been successfully sent, the follower case $c'(p)$ will become $\{returnName, items\}$.

6. CONCLUSION

We have defined a 3-dimensional net-based model for object-oriented systems. It maps object-oriented concepts such as messages and inheritance to predicates and methods to transitions and net refinements. In particular, the representations for inheritance and messages are unified. In other words, they are modelled by the same notion in net theory. In terms of graphic outlook, NOODLE is different from other models in that it is a 3-dimensional model. A CASE tool can be built so that the model can be manipulated and visualized easily by showing only components on selected 2-dimensional planes.

We propose to further our research by implementing the proposal in a development environment which incorporates NOODLE into popular object-oriented analysis and design methodologies and helps to solve problems which arise during the system development process.

REFERENCES

[1] A.J. Alencar and J.A. Goguen, "OOZE: an object-oriented Z environment", in *Object-Oriented Programming: Proceedings of the 5th European Conference* (*ECOOP 1991*), P. America (ed.), Lecture Notes in Computer Science, vol. 512, Springer, Berlin, Germany, pp. 180−199 (1991).

[2] M. Baldassari and G. Bruno, "An environment for object-oriented conceptual programming based on PROT nets", in *Advances in Petri Nets 1988: Proceedings of the 8th European Conference on Applications and Theory of Petri Nets*, R. Rozenberg (ed.), Lecture Notes in Computer Science, vol. 340, Springer, Berlin, Germany, pp. 1−19 (1988).

[3] D. Bjorner and L. Druffel, "Position statement", Workshop on Industrial Experience Using Formal Methods, Nice, France (1990). Also in *Proceedings of the 12th IEEE International Conference on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, pp. 264−266 (1990).

[4] M. Blaha and W. Premerlani, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, Englewood Cliffs, NJ (1998).

[5] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, CA (1994).

[6] G. Bruno and A. Balsamo, "Petri net-based object-oriented modeling of distributed systems", in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA 1986*), *ACM SIGPLAN Notices* **21** (11): 284−293 (1986).

[7] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ (1991).

[8] P. Coad and E. Yourdon, *Object-Oriented Design*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ (1991).

[9] T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ (1979).

[10] H.J. Genrich, "Predicate/transition nets", in *Advances in Petri Nets 1986*, Part 1: *Petri Nets, Central Models, and their Properties*, Lecture Notes in Computer Science and W. Brauer, W. Reisig, and R. Rozenberg (eds.), vol. 254, Springer, Berlin, Germany, pp. 207−247 (1987).

[11] J.A. Goguen and J. Meseguer, "Unifying functional, object-oriented, and relational programming with logical semantics", in *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (eds.), MIT Press, Cambridge, MA, pp. 417−477 (1987).

[12] C.A. Lakos and C.D. Keen, "LOOPN++: a new language for object-oriented Petri nets", in *Proceedings of the 1994 European Simulation Multiconference*, Elsevier, Amsterdam, The Netherlands (1994).

[13] J. Martin, *An Information Systems Manifesto*, Prentice Hall, Englewood Cliffs, NJ (1984).

[14] M. Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ (1988).

[15] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, NJ (1981).

[16] W. Reisig, *Petri Nets: an Introduction*, EATCS Monographs on Theoretical Computer Science, vol. 4, Springer, Berlin, Germany (1985).

[17] S.A. Schumann, D.H. Pitt, and P.J. Byers, "Object-oriented process specification", in *Specification and Verification of Concurrent Systems*, C. Rattray (ed.), Workshops in Computing, Springer, Berlin, Germany, pp. 21−70 (1990).

[18] E.V. Seidewitz, "General object-oriented software development: background and experience", *Journal of Systems and Software* **9** (2): 95−108 (1989).

[19] T.H. Tse, *A Unifying Framework for Structured Analysis and Design Models: an Approach Using Initial Algebra Semantics and Category Theory*, Cambridge Tracts in Theoretical Computer Science, vol. 11, Cambridge University Press, Cambridge. Hardback edition (1991). Paperback edition (2009).

[20] A.I. Wasserman, P.A. Pircher, and R.J. Muller, "The object-oriented structured design notation for software design representation", *IEEE Computer* **23** (3): 50−63 (1990).

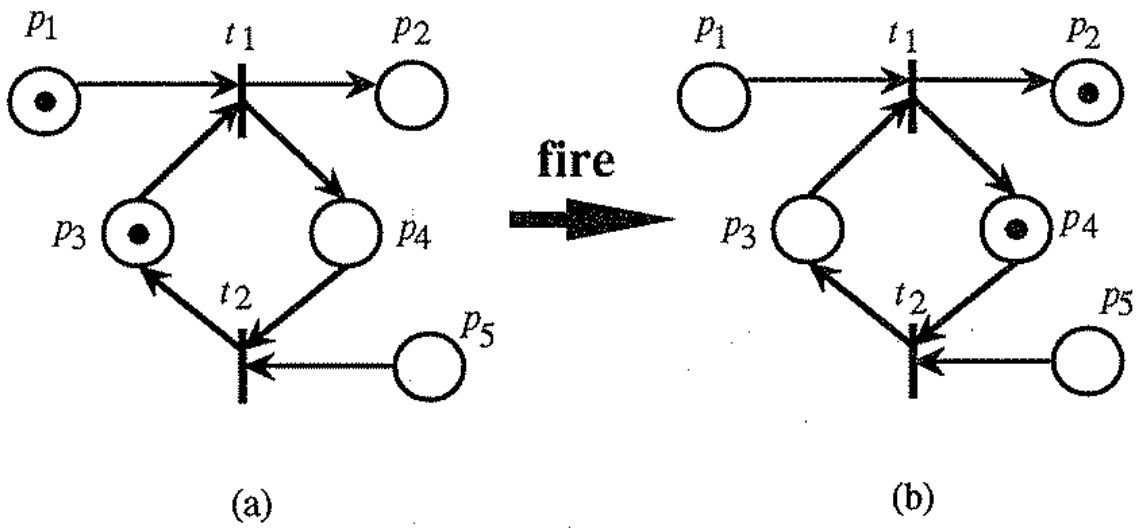

Figure 1 An example of the firing of a Petri net

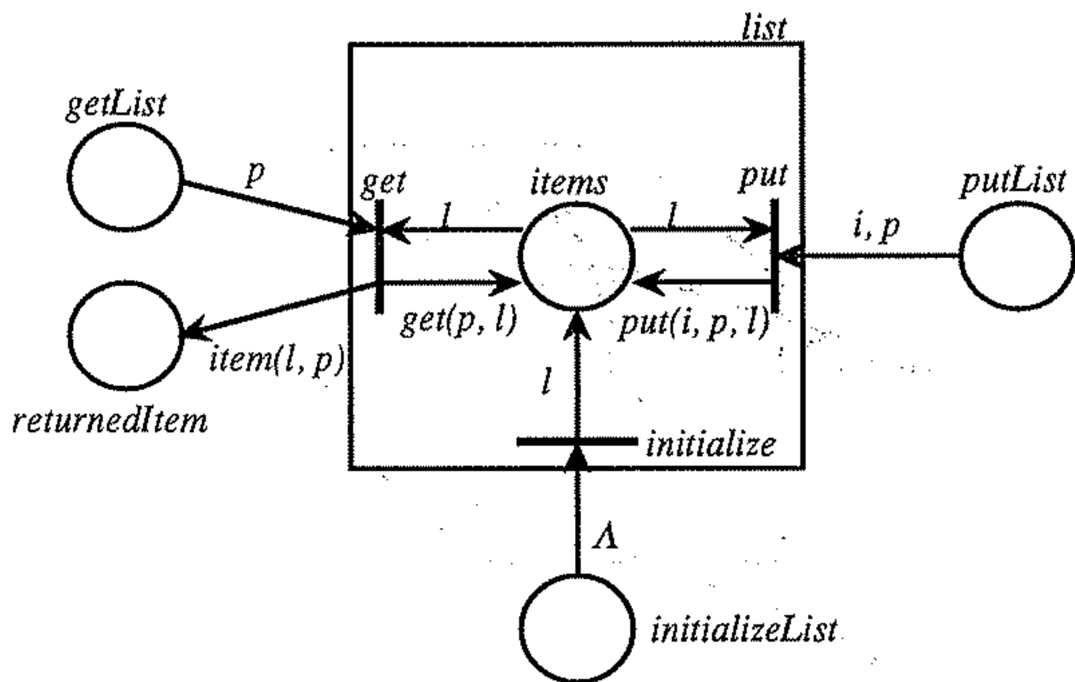

Figure 2 Graphic representation of the class *list*

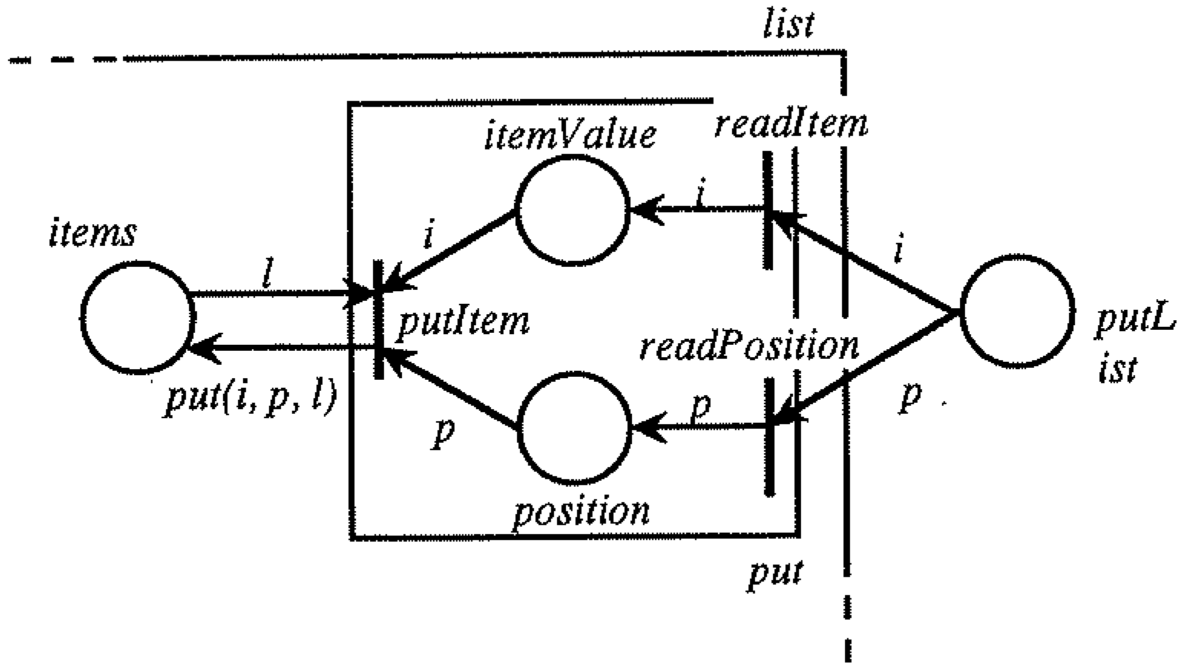

Figure 3 Graphic representation of the refinement of the method *put* of the class *list*

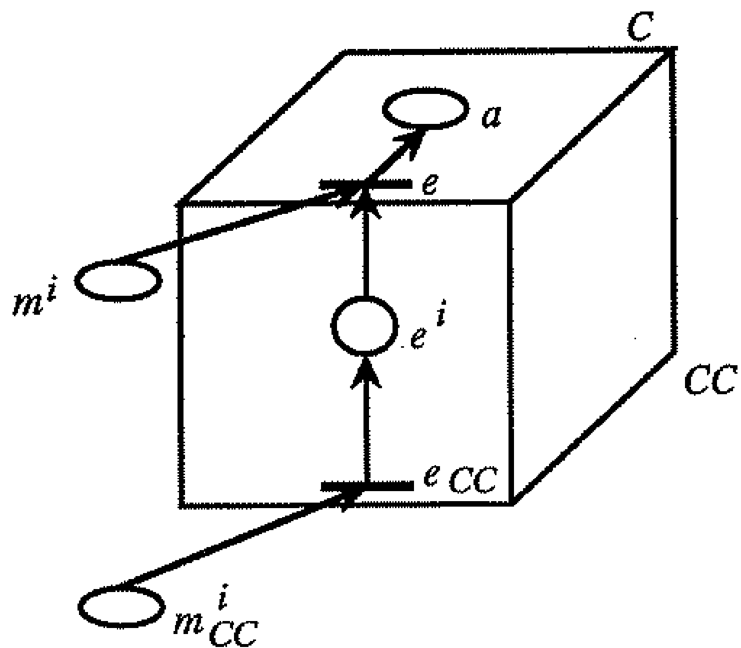

Figure 4 An inherited method which only accepts input messages

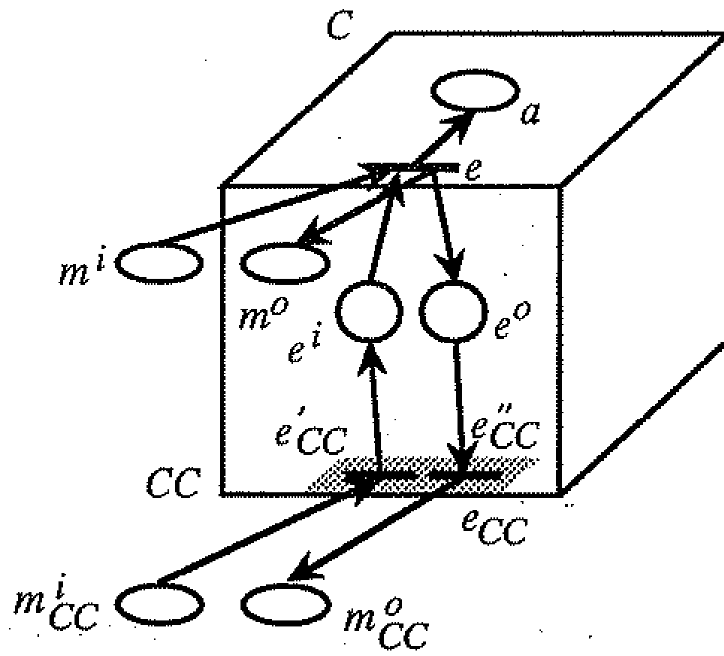

Figure 5 An inherited method which accepts input messages and returns messages

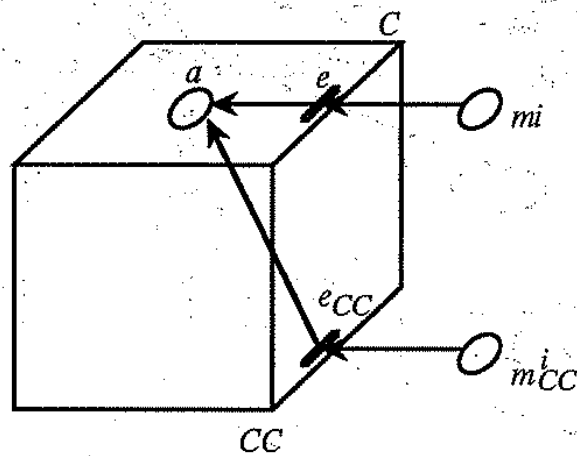

Figure 6 An overloaded method which only accepts input messages

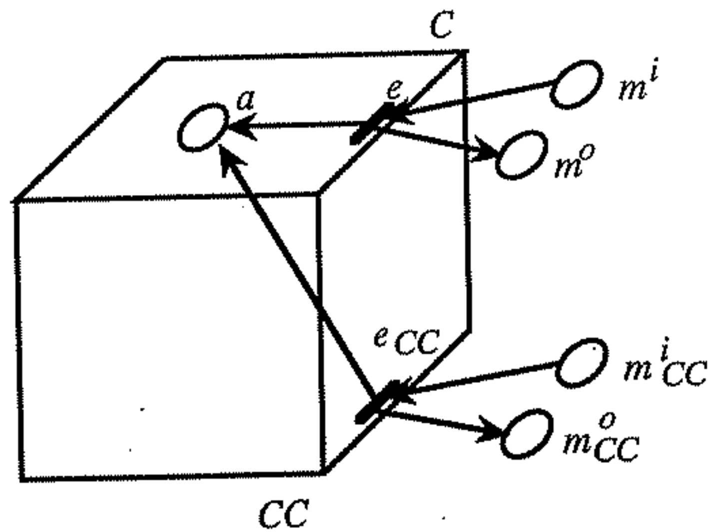

Figure 7 An overloaded method which accepts input messages and returns messages

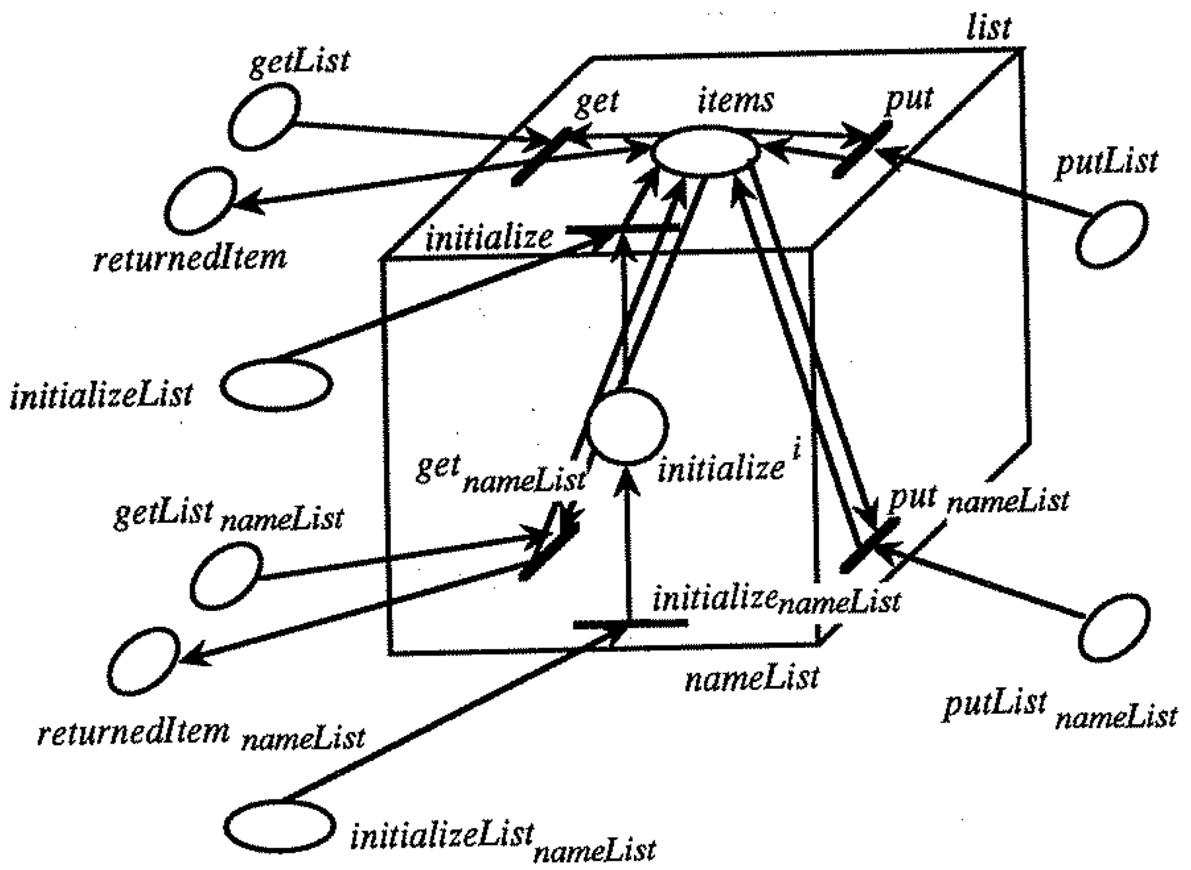

Figure 8 Graphical representation of the classes *list* and *namelist*

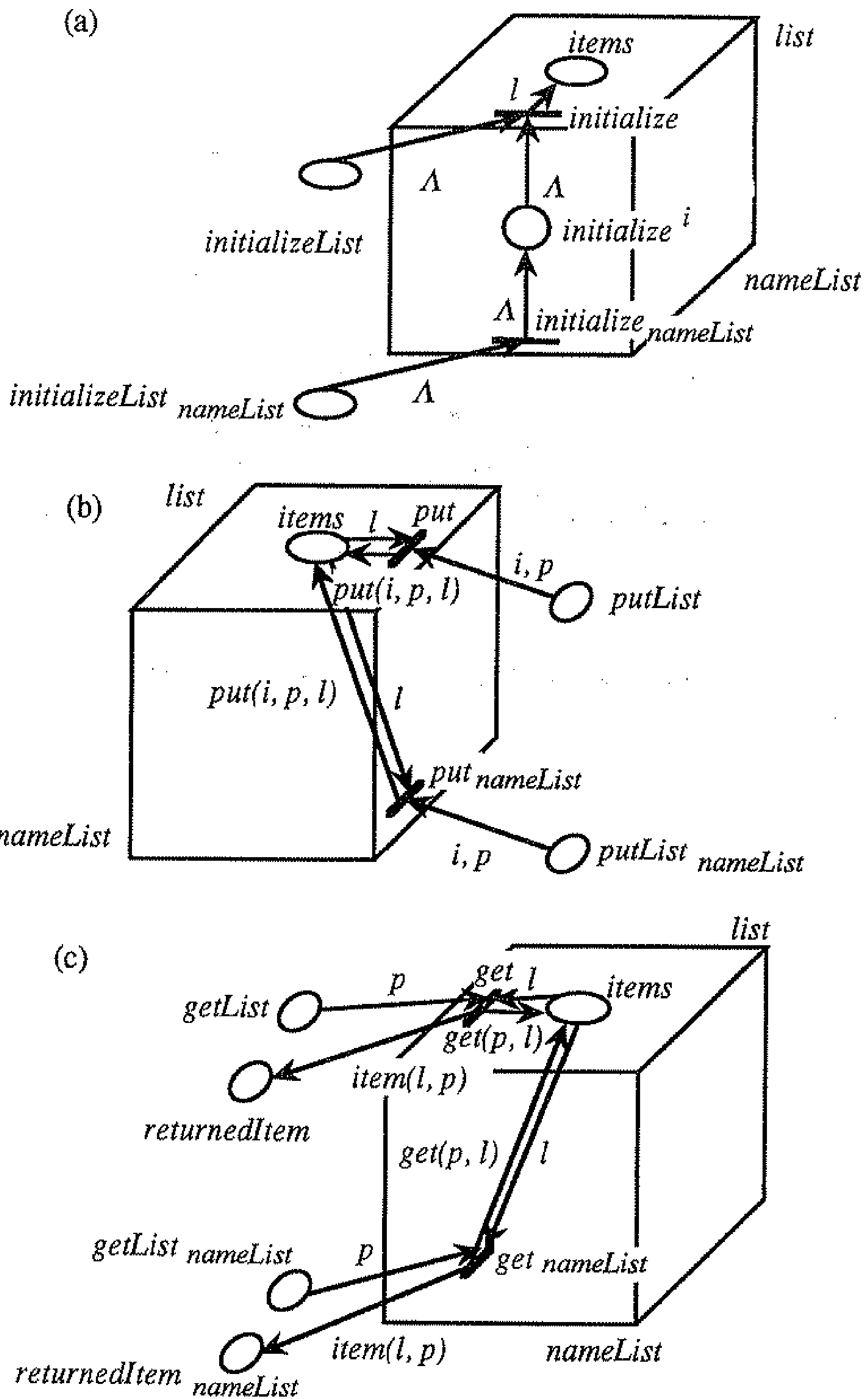

Figure 9 Graphic representations of the methods of the class *list*

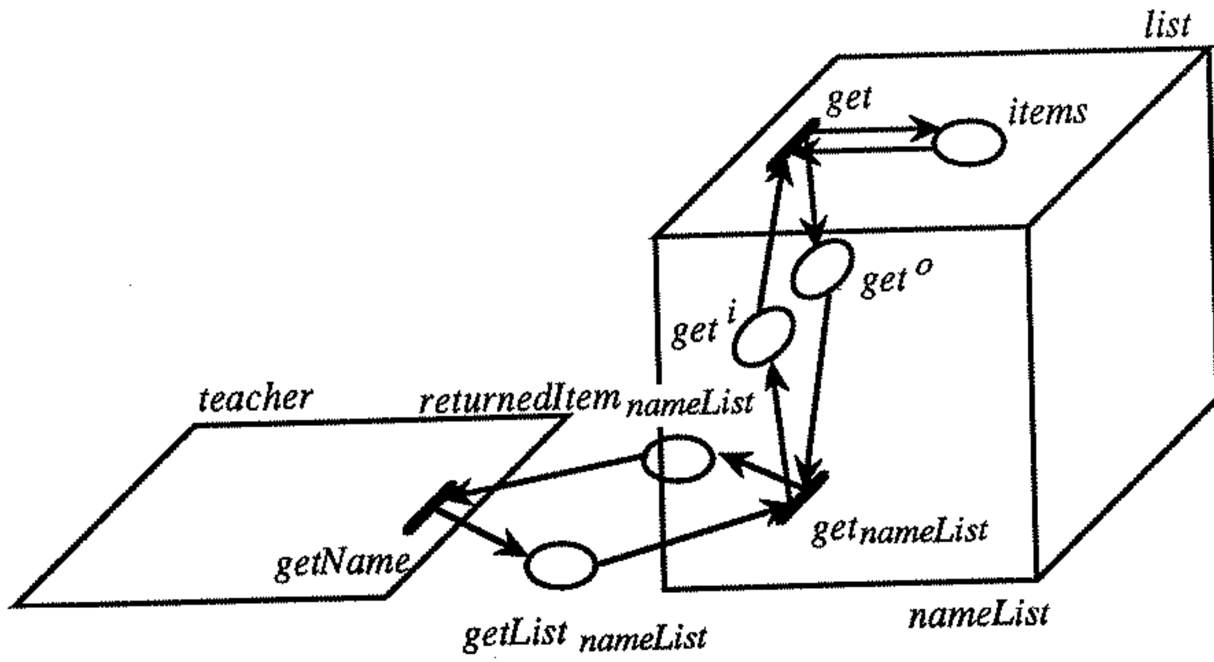

Figure 10 Graphic representation of message passing