An Algorithm To Manage Multiple Inheritance in An Object-Oriented Software Engineering Tool

Franck Barbier

IRIN (Computer Science Research Institute of Nantes)
University of Nantes
2, rue de la Houssinière
44072 Nantes CEDEX 03
FRANCE
e-mail: Franck.Barbier@irin.univ-nantes.fr

Abstract: Object-oriented methods lead to the construction of models —in other words, ensure the formal definition of requirements— in which inheritance relationships, sometimes multiple inheritance relationships, are depicted. A technique for validating models using multiple inheritance consists of defining, during the early activity of software development, precise semantics of inheritance to estimate appropriateness and relevance of the models constructed: to understand them, to make them conform to requirements, to validate them in order to pursue software development up to implementation. This technique is automated with the help of an algorithm implemented into a software engineering tool, which is now operational.

Keywords: object-oriented methods, object-oriented software engineering tools, multiple inheritance, requirements engineering, model verification.

1. Introduction

In the realm of software engineering, inheritance is often known as an object-oriented languages programming technique which ensures software reusability. In the realm of artificial intelligence, inheritance is known as a knowledge representation technique. When used in object-oriented methods, inheritance encompasses both of these aspects because besides the need for reusability, formalizing requirements cannot be ignored.

Within an industrial context, use of object-oriented methods cannot take place without object-oriented software engineering tools as characterized in [Meyer, 1994]. Unfortunately, implementation of object-oriented software engineering tools is difficult to do due to the (often graphical) poor specification language of most current object-oriented methods. Therefore, tools cannot effectively assist software engineers during the formal definition of requirements and, in practice, implementation of object-oriented software engineering tools often distorts methods by "adapting" their graphical syntax, in other words, their description capability.

Inheritance is at the central point of this problem because most popular object-oriented methods advocate multiple inheritance. Let us consider a model which incorporates multiple inheritance relationships. What type of a verification can be performed to ensure consistency, and then correctness of the model; in other words, which is the validation technique that can contribute to rigorous model construction?

In order to answer this question, this article presents a reflection upon and solutions relevant to the use of multiple inheritance in object-oriented software development methods when supported by software engineering tools. This article first reviews the need for checking functions in object-oriented software engineering tools. Such checking functions are rather ill-defined if no precise semantics of inheritance and of additional notation is established. In this respect, the choice made in this article is the inheritance relationship as that of inclusion according to set theory.

Next, with the help of the above hypothesis and of an example, a general technique to validate a multiple inheritance graph is empirically described. In the proposed logic, the initial multiple inheritance graph is transformed into an equivalent single inheritance graph. This transformation is possible thanks to the use of additional notation in the model: as it happens, exclusion and totality constraints concerning object types linked to one given object type. Then, the technique is accurately described with the help of an algorithm.

Before closing, this article will point out some of the advantages as well as disadvantages of the proposed technique, such as combinatorial problems, precisions concerning the algorithm's functioning and several implementation strategies based on various development systems (object-oriented programming languages with multiple inheritance or not, relational database management systems, etc.).

2. Subject Area

2.1 Theme

The wide and systematic use of object-oriented methods in industry leads to the implementation of these methods into object-oriented software engineering tools. Object-oriented software engineering tools essentially provide supports to construct object-oriented models based on object-oriented methods and their associated graphical notation. Roughly speaking, the construction of object-oriented models —with or without the help of object-oriented software engineering tools— generates two activities. The first is the formalization of the requirements. The second is the verification of the requirements. Note that the term "verification" means «The process of evaluating the products of a given software development activity to determine correctness and consistency with respect to the products and standards provided as input to that activity.» (DOD-STD-2167A, 1988).

Formal means "leading to a mathematical interpretation of models". Therefore, most popular object-oriented methods and their associated informal notation often limit model verification capability.

In this respect, this article focuses on the activity of object-oriented model verification — especially, automatic or semiautomatic verification while emphasizing the implementation of object-oriented methods into object-oriented software engineering tools— and focuses on inheritance —multiple inheritance in particular— because inheritance is a fundamental concept of object-orientation and is widely held in object-oriented models.

2.2 Implementation of Object-oriented Software Engineering Tools

Building object-oriented representations is often limited to the mastery of a graphical syntax. For this reason, object-oriented software engineering tools are above all editors dedicated to such a graphical syntax. This purpose is necessary but greatly insufficient. The semantics

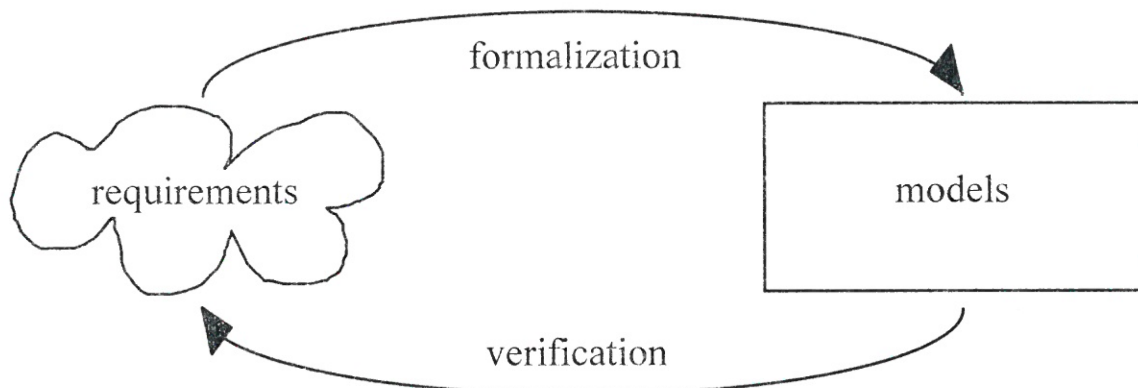


Figure 1. Model Construction = Requirements Formalization + Model Verification

In this line of reasoning, models, which are formalized requirements, express software goals with more precision than sentences in natural language. As for verification, in such a context, it precisely consists of the process of evaluating the models resulting from the requirements formalization activity to determine correctness and consistency with respect to the requirements in natural language provided as input to that activity.

However, if formalizing requirements improves precision so that elimination of ambiguity, inconsistency and incompleteness increases [Davis, 1988], use of graphical notation cannot ensure at all the same deep and robust model verification as use of formal notation can do.

of object-oriented representations is often not properly taken into account due to the lack of formal bases of most current object-oriented methods. In fact, a major expectation of a software engineer who uses an object-oriented software engineering tool is not only editing functions but checking functions of the representations built. To implement such checking functions, verification algorithms have to be defined and the formal nature of a method greatly helps this definition.

2.2.1 Related Experience

This remark is based on practical experience in implementing the object-oriented software

engineering tool [Barbier and Reich,1997] which supports the theoretical results presented in this article. A main criticism of Version 1.0 of this software has been the absence of "evaluation" of inheritance graphs, when built with multiple inheritance relationships and with another concept called "overlapping generalization" in the OMT method [Rumbaugh et al, 1991], "disjoint and non-disjoint sub-types" in the Syntropy method [Cook and Daniels, 1994] or "selective multiple inheritance" in [Dori and Tatcher, 1994]. The algorithm detailed in this article is the result of the search for a general technique to manage multiple inheritance in object-oriented software engineering tools. "general" means that the technique is free from a given method and established with an extension and a precise definition of the concept above, afterwards called "exclusion constraint". This technique is fully described in this article and is now operational in Version 2.0 or later of the software. Since this software was written in Smalltalk-80 [Goldberg and Robson, 1983], a small section of code of the algorithm is presented in Annex. The aim of this Section of code is just to clarify and improve the understanding of a central point of the proposed algorithm.

2.2.2 Problem Resolution Strategy

Before becoming a problem of tool implementation, the problem rests above all of methods. A prevailing trend is to create formal object-oriented methods, consequently new notation, to facilitate the implementation of these methods. The Room method [Selic, Gullekson and Ward, 1994] or object-oriented modeling with statecharts [Harel and Gery, 1997], which both focus on dynamic requirement aspects and on model verification based on model executability as characterized in [Blumofe and Hecht, 1988] or in [Barbier, 1992], are archetypes. Another prevailing trend is to expand informal object-oriented methods instead of creating formal ones. This is the case for example in [Cook and Daniels, 1994], in [André et al, 1995] or in [Bourdeau and Cheng, 1995] concerning the improvement of the OMT notation. The approach of Cooks and Daniels is inspired by Z to model static requirement aspects and by statecharts to model dynamic requirement aspects. The approach of André et al mixes up algebraic specifications and statecharts to model dynamic requirement aspects, and ignores static requirement aspects. The approach of Bourdeau and Cheng is taken from algebraic specifications to model static requirement aspects and ignores dynamic requirement aspects.

Concerning the second trend. the difficulty is the choice of an appropriate mathematical theory on which lies the formal notation. Type theory, for example, seems a promising way to follow. However, in industry, few methods, and thus, few software engineering tools, support type theories while incorporating object-orientation, these theories being too complicated, heavy, and above all not stabilized [Danforth and Tomlinson, 1988]. The recent results concerning the most widespread object-oriented methods [Hutt, 1994a] [Hutt, 1994b] confirm the opinion above —for example, none of these methods supports fully and efficiently a given type theory. These results describe and compare methods like the Objectory method [Jacobson et al, 1992] illustrating what is regarded as a non formal method or the Demeter method [Lieberherr, 1996], once more an archetype of what really is a formal object-oriented method.

2.3 Semantics of Inheritance in Object-oriented Methods

Inheritance is rather known as a programming language technique. Therefore, most research results have been obtained in this area and synthesized in [Ducournau and Habib, 1989][1] and [Ducournau et al, 1995]. In object-oriented methods, the expression "generalization/specialization" is used instead of "inheritance", or simply "specialization" in [Hutt, 1994b] in which it is defined as «Specialization: which occurs when an object type inherits operations, attribute types and relationship types from one or more super-types (with possible restrictions).».

2.3.1 Minimal Semantics

Use of inheritance in object-oriented methods is very intuitive. Contrary to programming languages, methods serve conceptualization concerns rather than implementation concerns —within the classification proposed in [Lalonde and Pugh, 1991], conceptualization concerns can freely correspond to the "is-a" relationship or to the "subtyping" one, rather than to the "subclassing" one which is programming languages-dependent. In this respect, as observed in [Taivalsaari, 1996], one cannot emphasize this strongly enough: «On the basis

[1] an entire formalization of multiple inheritance can be found in this article.

of the preceding discussion it appears that the analogy between inheritance and conceptual specialization is a lot weaker than has often been claimed.». According to this remark, a commonly accepted semantics compatible with almost all the methods has to be found, these methods considering inheritance either as a static mechanism of representation or as a support for polymorphism. In this sense, the semantics of the inheritance relationship considered as the inclusion relationship, is chosen. This point of view is the same as that found in [Atzeni and Parker, 1988]: «(...) an is-a can be considered as an integrity constraint, requiring the set of instances of a type to be subset of another.». As a result, this leads to a mathematical interpretation of inheritance as follows[2]:

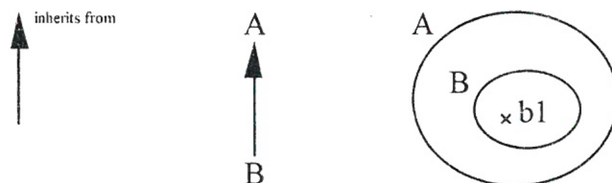


Figure 2. If object type B inherits from object type A then an object b1 belonging to set B belongs to set A because B is included in A

In the wake of object-oriented programming languages, object-oriented methods support multiple inheritance relationships between object types, such as:

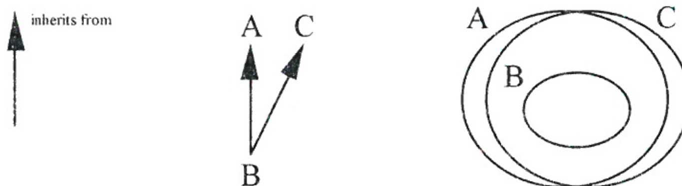


Figure 3

By their very nature, multiple inheritance relationships are the source of many conflicts and consequent ambiguities. For example, if p is a property of A and C, what is the status of B in relation to p? In object-oriented programming languages, the ambiguities are eliminated with the help of various complex algorithms supported by graph theory [Demichiel, 1988] [Ducournau and Habib, 1989] [Ducournau et al, 1995]. "various" means that each algorithm is strongly linked to a given programming language. Though these verification algorithms may solve the problems, and especially programming problems, they are not easily understood by the average software engineer because they do not rest on a common uniform interpretation of inheritance in programming languages.

In object-oriented methods, the use of inheritance is, contrary to what is found in object-oriented programming languages, seldom rigorous. In this context, the use of multiple inheritance in object-oriented methods is beneficial if the resulting description of the requirements in question does not hinder the understanding, and thus the verification, of the models which represent these requirements. This point becomes more and more involved when using additional notation to increase description capability of a method. This additional notation consists of the concept of "exclusion constraint" and the concept of "abstract" object type.

2.3.2 Modeling Concepts Linked To the Inheritance Concept

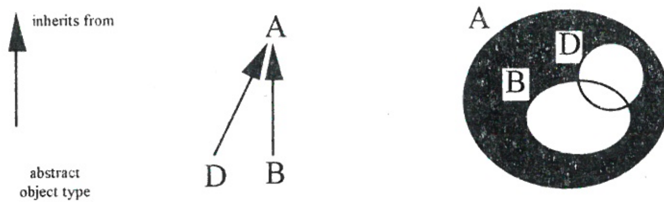
Once more, use of this additional notation is awkward if it is not associated with precise semantics, especially with that of the inheritance relationship as an inclusion relationship.

Let us firstly consider the concept of "abstract" object type. If this concept is clear in object-oriented programming languages, this is not the case in object-oriented methods. For example in Eiffel [Meyer, 1992], the term "deferred" is used in place of the term "abstract" with this definition: «A class which has one or more deferred routines is itself said to be deferred». The advantage of giving semantics to inheritance, when it is used with conceptualization concerns, leads to the unique mathematical interpretation of the concept of "abstract" object type in Figure 4.

This mathematical interpretation is not innovative and known for example as a "constraint of totality between two or more object types which directly inherit from one given object type" in the NIAM (Nijssen Information Analysis Method) method [Habrias, 1988].

Let us secondly consider the concept of "exclusion constraint" which is not innovative either and is used once again in the NIAM method (it is known as a "constraint of

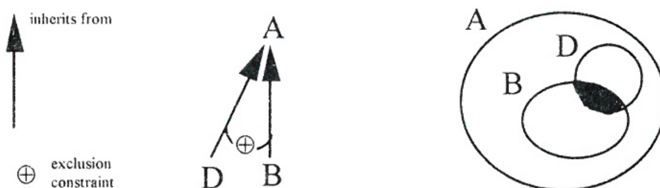
[2] the Venn diagram is a useful widespread technique to depict such a mathematical interpretation.



: region indicating the empty set (i.e. Ø)

Figure 4. There is no object which belongs to set A which does not belong to set B or (not exclusively) set D

exclusion between two or more object types which directly inherit from one given object type") as well as in other approaches concerning knowledge representation such as that presented in [Atzeni and Parker, 1988] or that presented in [Agopian, 1992].



: region indicating the empty set (i.e. Ø)

Figure 5. If an object belongs to set B then it does not belong to set D (and vice versa)

2.3.3 Extended Semantics

More advanced mathematical interpretations of inheritance exist, like Bourdeau's and Cheng's approach, where "subtyping" is defined as a total function from a subtype to its unique direct super-type. Although this approach avoids "multiple subtyping", the proposed semantics is different —and in this respect, rather original— from many other recognized and widely held approaches, especially Cooks' and Daniels' or Shlaer's and Mellor's [Shlaer and Mellor, 1992], where semantics of inheritance is defined with the help of relationships between finite state machines.

Considering almost all the methods, minimal semantics of inheritance is the same as that chosen in this article (see for example, [Rumbaugh et al, 1991, pp.65-69] or [Cooks and Daniels, 1994, pp.67-70]. However, extended semantics of inheritance, if it exists, is not the same from one method to another. It is therefore difficult to be free from one method when implementing an object-oriented software engineering tool. So, the problem becomes the same as that of the object-oriented programming languages and is known as the problem of "methods interoperability". In this context, this article does not explore deep semantics of inheritance to make the proposed results methods-independent.

2.4 Verification

As written above, verification is the general process of evaluating the products of a given software development activity. In this sense, test is a kind of verification. This article focuses on one kind of verification, that of the early activity of software development, often called "Object-Oriented Analysis" in many object-oriented methods [Shlaer and Mellor, 1988] [Coad and Yourdon, 1991a] [Rumbaugh et al, 1991] or called "Software Requirements Analysis" in [DOD-STD-2167A, 1988] or "Object Analysis Modelling" in [Hutt, 1994b] according to whom «The purpose of [object] analysis modelling is to obtain a thorough description of a problem domain, so that the requirements for applications supporting the domain can be formalized and the environment in which those applications are to be used is well understood.».

In this perspective, verification determines whether object-oriented models are consistent, and then correct, according to two criteria:

- is an object-oriented model self-consistent? For example within an object-oriented model, if an object type A inherits from an object type B and B inherits from A, there exists an error which can be detected independently of the input requirements. Object-oriented software engineering tools have to deal with these "first degree" verifications like avoiding cycles between object types linked by inheritance relationships or like avoiding repeated inheritance (an example of repeated inheritance is shown and explained in Figure 7).
- is an object-oriented model a correct model of a list of requirements? This implies that verification is the process of comparing a list of requirements in

natural language to an object-oriented model which is its formalization. These "second degree" verifications are complicated, or even undefined because based on natural language processing. Furthermore, verification does not only address inheritance and the list of verifications which have to be implemented into object-oriented software engineering tools is varied and sizeable. Current research results in natural language processing are not sufficiently advanced given the issue at stake.

Faced with the second criterion, the main idea of this article is to transform models incorporating multiple inheritance relationships so that the verification process is simplified and effectively assisted by the tool. The proposed technique follows the same idea as that of the Demeter method: «However, we don't intend for our algorithms be used to restructure class hierarchies [inheritance graphs] without human control. We believe that the output of our algorithms makes valuable proposals to the human designer who then makes a final decision» [Lieberherr, 1996]. Thus, this technique works without natural language processing on the assumption that a model which has multiple inheritance relationships is difficult to understand and hence difficult to validate by a software engineer, while a model which has single inheritance relationships is not subject to these limitations. This assertion is empirically studied with the help of an example and can now be illustrated in Figure 6.

3. Intuitive Approach

The general goal of this Section is to illustrate the introduction of two distinct parts, afterwards called phases, in the proposed algorithm. An example helps us to achieve this goal.

3.1 Presentation of An Example

The following example is a multiple inheritance graph resulting from an analysis. This example will facilitate several tasks:

- explaining in greater detail the meaning of the constraints of exclusion and of totality;
- transforming empirically this multiple inheritance graph into a single inheritance graph;
- invalidating the multiple inheritance graph since, as asserted in this article, a single inheritance graph is its equivalent. Consequently, to invalidate the single inheritance graph is equivalent to invalidating the multiple inheritance graph.

One can criticize this model, in particular show the redundant relationship between "F=Spy helicopter" and "C=Military helicopter" (it is a typical example of repeated inheritance). This "first degree" criticism (in fact, as written above, made independently of the input

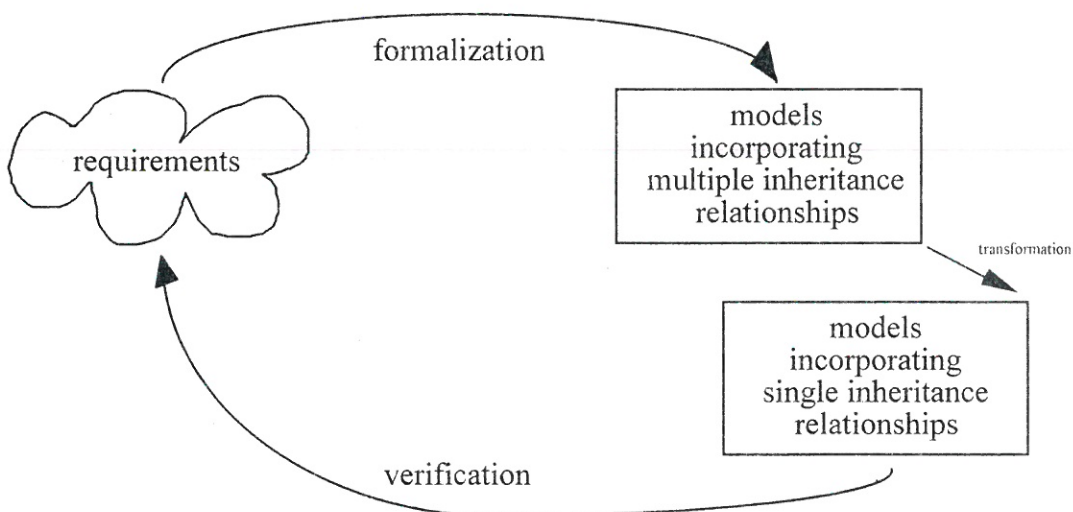


Figure 6

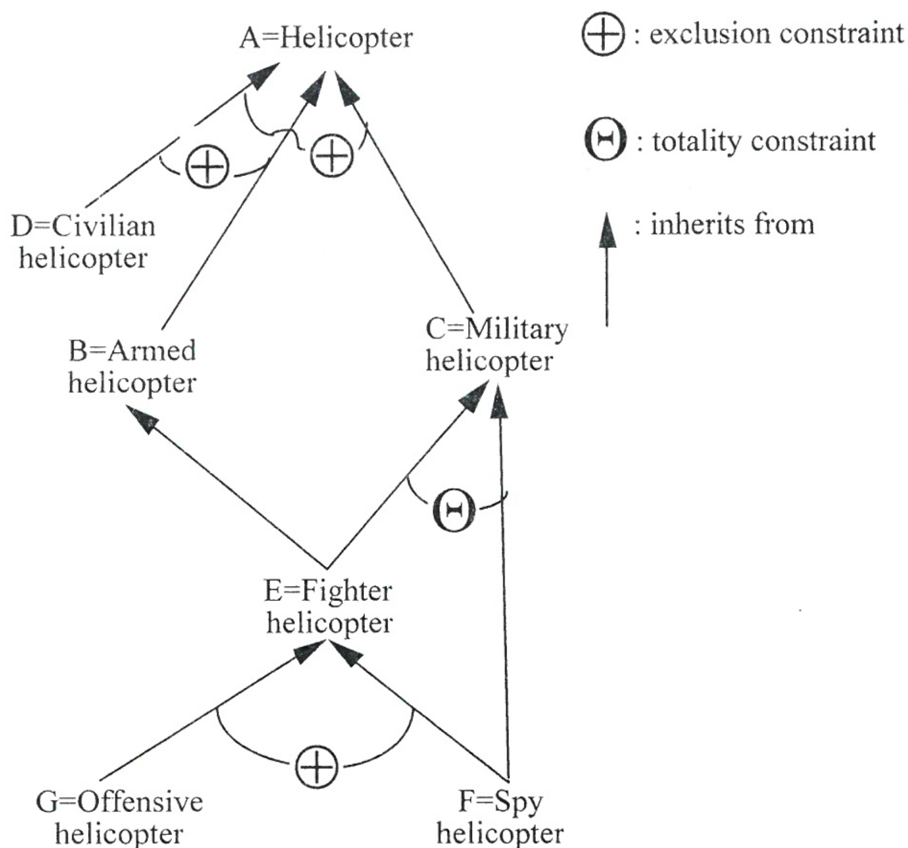


Figure 7

requirements) can be implemented as a warning into an object-oriented software engineering tool. However, a deeper study shows a totality constraint partly supported by this relationship between F and C and therefore a more precise study is necessary.

The model shown in Figure 7 shows object types linked by inheritance relationships. The ⊕ sign is a constraint of exclusion between object types which directly inherit from one common object type. It is important to note the fact that such a constraint cannot be expressed between object types which are not directly linked to one given object type. This latter object type is called the father object type, while those which are directly linked to it are called son object types. According to set theory, a constraint of exclusion between son object types means that their intersection is equal to the empty set.

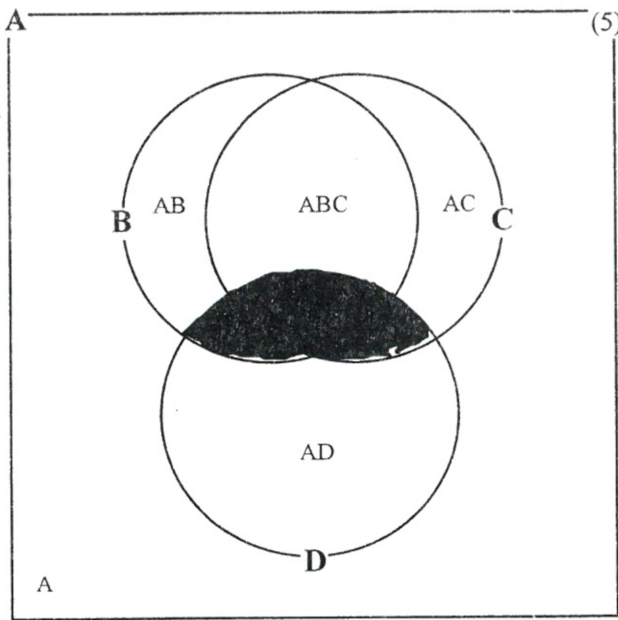
The sign Θ is a constraint of totality between the son object types of a given object type, subject to the same remarks as noted above. Again, according to set theory, a constraint of totality between son object types means that their union is equal to the set represented by the father object type.

A set theory interpretation of Figure 7 yields those presented in Figure 8.

There, one may observe that:

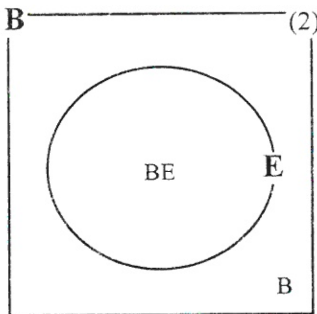
- the squares and the ellipses are sets associated with the object types within Figure 7. When an ellipse is located within a square, this indicates, as far as the inheritance relationship is considered as that of inclusion, an inclusion relationship between two sets. Thus, one may consider Figure 8 as a set theory interpretation of Figure 7.
- the black areas are regions eliminated after having satisfied the constraints of exclusion and of totality. In fact, these regions indicate the empty set (i.e. Ø);

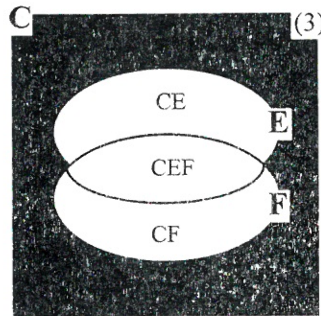
Before describing intuitively and formally the proposed verification technique and its associated algorithm, two points have to be emphasized.

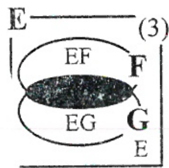


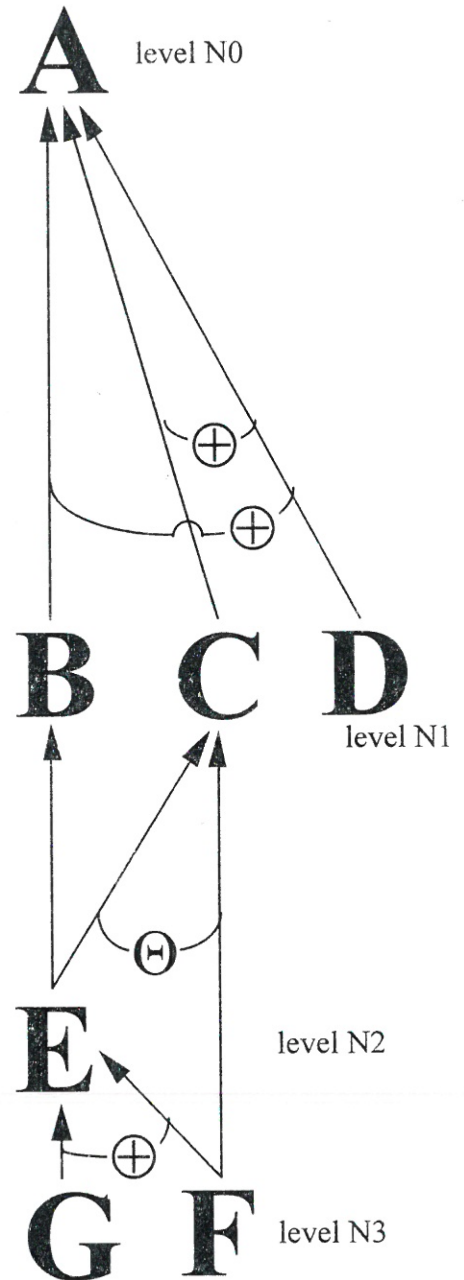
: region eliminated after having satisfied the constraints of exclusion and of totality (i.e. region indicating the empty set: Ø)

between () : number of regions remaining after having satisfied the constraints of exclusion and of totality (i.e. r')









set theory interpretation of initial inheritance graph

initial inheritance graph (figure 7)

Figure 8

1) According to [Ducournau and Habib, 1989], an inheritance graph is an oriented graph without cycle $G_H=(N, H, A)$ with a root A. N is a set of object types derived from the object type A, called the maximum object type in relation to the inheritance relationship H. An

object type n_a is a father object type for an object type n_b if $[n_b n_a] \in H$. An object type n_b is a son object type for an object type n_a if $[n_b n_a] \in H$.

A graph without cycle can be ordered with the help of an algorithm which defines a relationship of equivalence for N and which then breaks down N into equivalence classes called levels.

Level 0 is defined by $N0=\{n \in N / \Gamma_H(n)=\varnothing\}$. Γ_H is a mapping from N to P(N) (powerset) such that $n' \in \Gamma_H(n)$ if $[nn'] \in H$. In this respect, $\Gamma^1 H(n) = \Gamma_H(n)$, $\Gamma^2 H(n) = \Gamma_H(\Gamma^1 H(n)) = \bigcup_{n' \in \Gamma^1_{H(n)}} \Gamma_{H(n')}, \ldots, \Gamma^k_{H(n)} =$

$$\Gamma_H(\Gamma^{k-1} H(n)) = \bigcup_{n' \in \Gamma^{k-1}_{H(n)}} \Gamma_{H(n')}$$

and finally $\Gamma^* H(n) = \bigcup_{k=1}^{|N|} \Gamma^k H(n)$ where |N| is the cardinal of N. Level i when i>0 is then defined by

$$N_i = \left\{ n \in N - \bigcup_{j=0}^{i-1} N_j \;/\; \bigcup_{j=0}^{i-1} N_j \supseteq \Gamma^* H(n) \right\}.$$

Formally, $\Gamma^* H(n)$ is the transitive closure of n without {n}. Intuitively, $\Gamma^* H(n)$ is a subset of N made up of attainable object types from n according to H. By definition, an inheritance graph $G_H=(N, H, A)$ is such that $N0=\{A\}$ and $\forall n \in N, n \notin \Gamma^* H(n)$. For example, in Figure 7, $\Gamma_H(F)=\{C, E\}$ and $\Gamma^* H(F)=\{A, B, C, E\}$.

2) Let $n \in N$. If m is the number of n's son object types, then a set theory interpretation of the m inheritance relationships permits a graphical representation of r regions where $r=2^m$.

The expression of the constraints of exclusion and of totality among the m son object types of n leads to the introduction of r' such that r'<r: r' regions should remain after the elimination of regions following the satisfaction of constraints.

For example, in the model shown in Figure 8, object type A has 3 son object types: B, C, and D, thus r=8. The constraint $B \oplus D$ implies the elimination of the regions associated with $B \cap D$ since $B \cap D = \varnothing$, and $B \cap C \cap D$ since $B \cap D \supset B \cap C \cap D$. The constraint $C \oplus D$ implies the elimination of the regions associated with $C \cap D$, and $B \cap C \cap D$ which is already eliminated, thus r'=5.

Henceforth, regions are named by the concatenation of the object types which they are composed of. For example, at level N2 of Figure 8, the remaining regions are E, EF, and EG (the region EFG has disappeared since $F \oplus G$). In region E, the instances of object types belong to set E. In region EF, the instances of object types belong to both set E and set F, etc.

These two points having been taken into consideration, it is now possible to comment in detail, for each level and each of the levels' object types, upon the regions which remain after the satisfaction of the constraints of exclusion and of totality.

When passing from a level Ni to a level Ni−1 when i=max to 1 (max is the index of the last level Nmax after ordering the graph), it is possible for each object type of the level Ni−1 to take into account the constraints associated with its son object types. For example, at level N2 of Figure 8, $E \supset F$, which implies that the complementary set of E in $E \cup F$ is equal to the empty set. This is not the case at level N1 of the same Figure, the complementary set of E in $E \cup F$ is not equal to the empty set, but to the region CF.

In order to eliminate this contradiction, the region CF of level N1 in Figure 8 must be eliminated. In general, a constraint satisfied for a son object type should also be applied to its father object types. According to this logic, Figure 9 presents the results for the example shown in Figure 8.

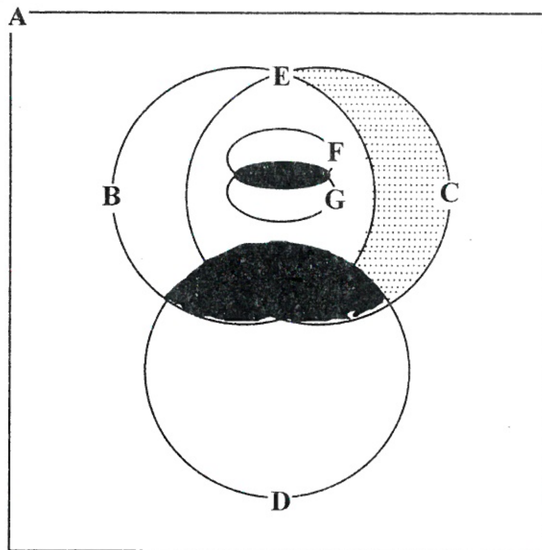
The final result of this process is the obtaining of a list of regions associated with level N0. In Figure 9, the list is as follows: A, AB, AD, ABCE, ABCEF, and ABCEG. This list enables one to sum up the whole of the constraints of exclusion and of totality of the model shown in Figure 8. Now, two questions arise: what does one do with this list? How does one automatically compute this list? The technique proposed in this article answers both these questions.

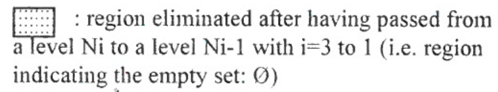
The posed hypothesis is that there exists an algorithm P1 (phase 1) which, when activated with as input a multiple inheritance graph, allows one to compute a list of regions associated with level N0; in other words associated with the root of the graph. Next, an ordering algorithm P2 (phase 2) is proposed with as input the list of regions associated with

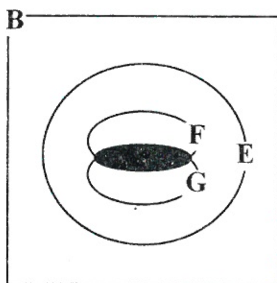
level N0, so as to obtain single inheritance graph(s) equivalent to the original multiple inheritance graph. Reasoning by analysis and synthesis leads first to the study of the feasibility of phase 2 (analysis) and then to the consideration of the feasibility of phase 1 (synthesis).

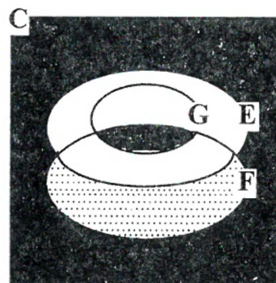
10, for the first iteration, n=A since A appears in 6 regions among 6; for the second iteration, n=B, etc. A, the root of the original multiple inheritance graph, is always the object type retained at the first iteration.

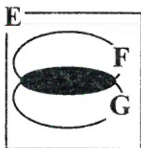
Phase 2 extracts from the list of regions











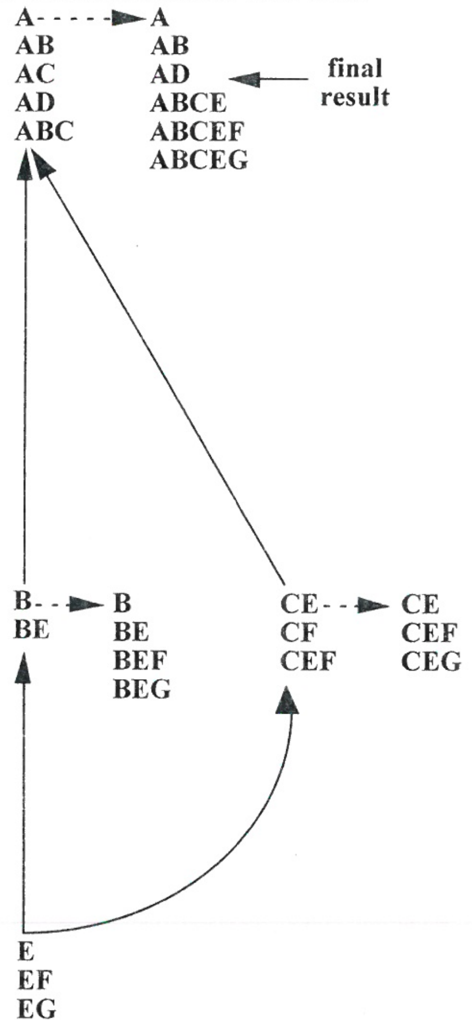


Figure 9

3.2 Phase 2 (P2)

The process of ordering each region of the list consists in retaining at each iteration, an object type n which remains from the previous iteration, such that n appears the most often in each region of the list. For example, in Figure

associated with level N0, M (M=2 in the example) single inheritance graphs equivalent to the original multiple inheritance graph according to the tree property defined in [Lieberherr, 1996]: «A collection of subsets of a set S has the **tree property** if for any pair of subsets of S one element of the pair is completely contained in the other, or if the two subsets are disjoint.». So, let Instances be a

mapping from L (L is the list of regions associated with level N0 and is formally defined in paragraph 3) to P(O) (O the set of constructable objects from the system modeled in Figure 7 and P(O) is the powerset), Instances(A) is the set of objects from type A, Instances(AB) is the set of objects from type A and type B, etc. Then the tree property is useful if and only if S=Instances(A) and Instances(A)$\supseteq$Instances(A), Instances(A)$\supset$Instances(AB), etc. In this logic, the following process is proposed:

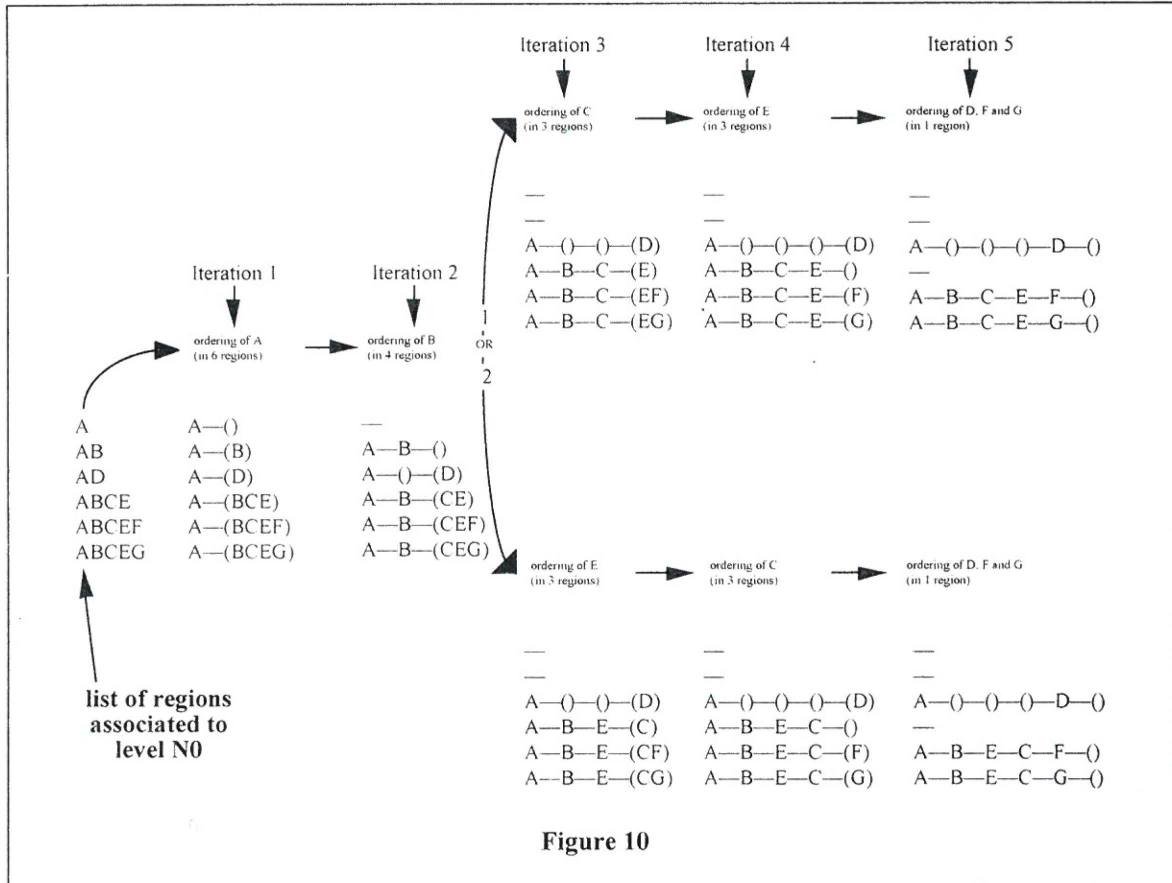


Figure 10

Intuitively, the object type retained at each iteration is the most "general" —according to the "generalization/specialization" logic— object type remaining from the previous iteration. More specifically, the ordering of each region of the list consists in enumerating one of several families of paths, a family <u>defining</u> a single inheritance graph which is equivalent to the original multiple inheritance graph.

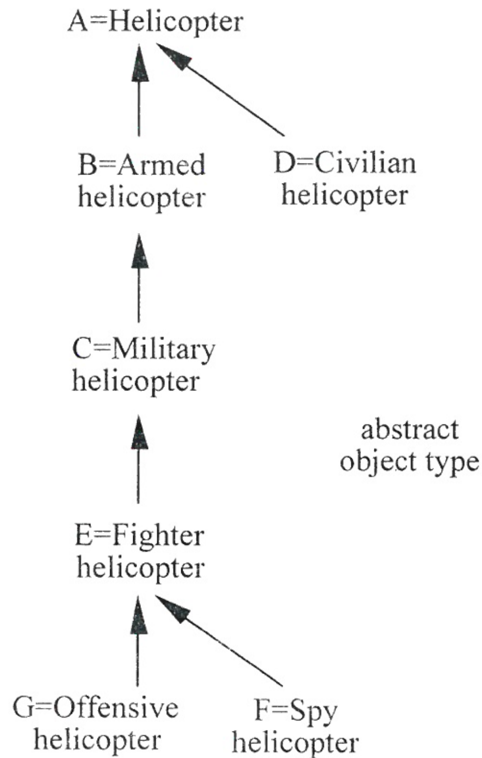
<u>Note</u>: an inheritance graph $G_{H'}$=(N', H', A') is a single inheritance graph if and only if, for all of n $\in$N' $-$ {A'}, there exists only <u>one</u> path between n and A' denoted n$\rightarrow_{H'}$ A'.

Let $G_{H'}$=(N', H', A') be a single inheritance graph. Let F(N') be a subset of N' such that F(N')={n $\in$ N' $-$ {A'} / ($\exists$ n' $\in$ N' $-$ {A'} / $\Gamma^*_{H'}$(n') $\cup$ {n'} $\supset\Gamma^*_{H'}$(n) $\cup$ {n}) $\Rightarrow$ n'=n}. The family of paths n $\rightarrow_{H'}$ A' such that n $\in$ F(N') defines the graph $G_{H'}$.

For example, in Figure 10 there are two single inheritance graphs which are equivalent to the original multiple inheritance graph. The first graph is defined by the following family of three paths: D $\rightarrow$A ([DA]), F $\rightarrow$ A ([FE], [EC], [CB], [BA]) and G $\rightarrow$A ([GE], [EC], [CB], [BA]). The second graph is defined by the following family of three paths: D $\rightarrow$ A ([DA]), F $\rightarrow$ A ([FC], [CE], [EB], [BA]) and G $\rightarrow$A ([GC], [CE], [EB], [BA]).

Hence, from the example shown in Figure 7, the two single inheritance graphs in Figure 11 have been obtained:

solution 1



solution 2

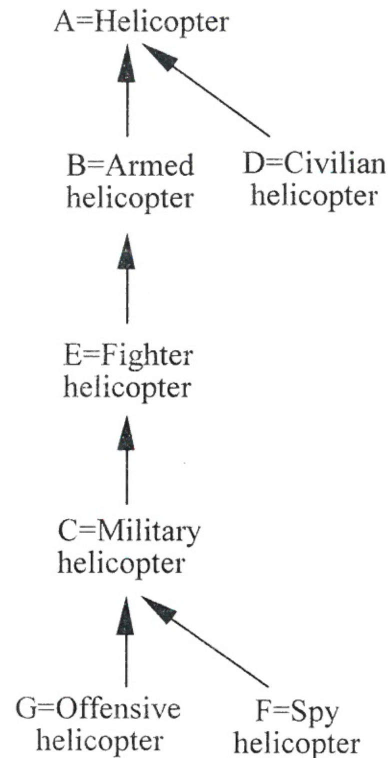


Figure 11

In the first solution, C is an abstract object type. In the second, E is an abstract object type. As a matter of fact, note that at iteration 3 of Figure 10, C has been ordered in the first solution, E in the second. What does the first solution show? That there is no instance of C which is not an instance of E (and vice versa in the second solution).

In studying solution 1 as well as solution 2 (Figure 11), one shall notice that a helicopter cannot be a military helicopter without being an armed helicopter which is discerned with difficulty in the model presented in Figure 7, because there is not at all a direct or indirect inheritance relationship from "Military helicopter" to "Armed helicopter"! Furthermore, let us imagine that the "Troop transportation helicopter" object type is directly linked to the "Military helicopter" object type in Figure 7. It surely means, according to Figure 11, that "Troop transportation helicopter" inherits from "Armed helicopter". How to detect this error in Figure 7 while it is easy to detect it in Figure 11?

To sum up, taking into account the comparison of the inheritance relationship to the inclusion one, and taking into account the fact that the multiple inheritance graph in Figure 7 is equivalent to the two single inheritance graphs in Figure 11, the model shown in Figure 7 is invalid.

3.3 Phase 1 (P1)

In order to come to this conclusion, it is assumed that there exists an algorithm P1 which allows one to compute the list of regions associated with the root of the multiple inheritance graph shown in Figure 7. Here is an empirical presentation of this algorithm P1, illustrated by a matrix representation (Figure 12): let M_A be a matrix associated to object type A, such that (on the line 0 of M_A are found r' regions resulting from the expression of the constraints of exclusion and of totality between B, C, and D; and in column 0 there are found the regions associated with B and C respectively, the result of taking into

consideration possible contradictions between levels 2 and 1 (D is absent from column 0 because D does not pose a problem insofar as it is not a father object type for any object types). In the matrix M_A, all of possible concatenations of regions are double-outlined. Algorithm P1, made up of three rules which will be presented, eliminates regions which are incompatible with the constraints already satisfied.

italics. This region is in fact inacceptable since it ignores at a level i–1, a constraint which was satisfied at a level i. Thus, the application of rule 1 yields those presented in Figure 13.

In M_C for example, line 3, column 2, CEFG is *in italics* since, in column 0 of M_C, the sequence EFG does not appear while it could have theoretically appeared through the

M_A	0	1	2	3	4	5
0	N1 to N0	**A**	**AB**	**AC**	**AD**	**ABC**
**						
1	B	*AB*	*AB*	*ABC*	**ABD**	*ABC*
2	BE	**ABE**	*ABE*	**ABCE**	**ABDE**	*ABCE*
3	BEF	**ABEF**	*ABEF*	**ABCEF**	**ABDEF**	*ABCEF*
4	BEG	**ABEG**	*ABEG*	**ABCEG**	**ABDEG**	*ABCEG*
**						
5	CE	**ACE**	*ABCE*	*ACE*	**ACDE**	*ABCE*
6	CEF	**ACEF**	*ABCEF*	*ACEF*	**ACDEF**	*ABCEF*
7	CEG	**ACEG**	*ABCEG*	*ACEG*	**ACDEG**	*ABCEG*

↑ ↑

M_B	0	1	2
0	N2 to N1	**B**	**BE**
******************************			
1	E	*BE*	*BE*
2	EF	**BEF**	*BEF*
3	EG	**BEG**	*BEG*

M_C	0	1	2	3
0	N2 to N1	**CE**	**CF**	**CEF**
**				
1	E	*CE*	*CEF*	*CEF*
2	EF	*CEF*	*CEF*	*CEF*
3	EG	**CEG**	**CEFG**	*CEFG*

in italics: redundant region with possible region
in bold-type: possible region

Figure 12

Note that in Figure 12 one must first compute the list of matrix M_B: B, BE, BEF, and BEG, as well as the list of matrix M_C: CE, CEF, and CEG before being able to compute the list of M_A. Indeed, B, BE, BEF, BEG and CE, CEF, CEG make up column 0 of M_A in Figure 12. In Figure 15, B, BE, BEF, BEG and CE, CEF, CEG are the final result within M_B and M_C.

Rule 1: if in a given matrix (here, the double-outlined area in M_A, M_B, or M_C), there is a region which includes a sequence of letters (i.e. a region which groups together a number of object types) that could have theoretically appeared in the 0 line or the 0 column of the matrix, the region is eliminated by putting it *in*

grouping of EF and EG.

Rule 2: if, in a given matrix, a region including a sequence of letters that is incompatible with a sequence of letters from the 0 line or the 0 column of the matrix, this region is eliminated by putting it *in italics*. As in rule 1, this region is inacceptable since it ignores at a level i–1, a constraint satisfied at a level i. The application of rule 2 yields those presented in Figure 14.

M_A	0	1	2	3	4	5
0	N1 to N0	**A**	**AB**	**AC**	**AD**	**ABC**
**						
1	B				*ABD*	
2	BE	**ABE**		**ABCE**	*ABDE*	
3	BEF	**ABEF**		**ABCEF**	*ABDEF*	
4	BEG	**ABEG**		**ABCEG**	*ABDEG*	
**						
5	CE	**ACE**			*ACDE*	
6	CEF	**ACEF**			*ACDEF*	
7	CEG	**ACEG**			*ACDEG*	

↑ ↑

M_B	0	1	2
0	N2 to N1	**B**	**BE**
*********************************			
1	E		
2	EF	**BEF**	
3	EG	**BEG**	

M_C	0	1	2	3
0	N2 to N1	**CE**	**CF**	**CEF**
**************************************				
1	E			
2	EF			
3	EG	**CEG**	*CEFG*	

Figure 13

M_A	0	1	2	3	4	5
0	N1 to N0	**A**	**AB**	**AC**	**AD**	**ABC**
**						
1	B					
2	BE	*ABE*		**ABCE**		
3	BEF	*ABEF*		**ABCEF**		
4	BEG	*ABEG*		**ABCEG**		
**						
5	CE	*ACE*				
6	CEF	*ACEF*				
7	CEG	*ACEG*				

↑ ↑

M_B	0	1	2
0	N2 to N1	**B**	**BE**
*********************************			
1	E		
2	EF	**BEF**	
3	EG	**BEG**	

M_C	0	1	2	3
0	N2 to N1	**CE**	**CF**	**CEF**
**				
1	E			
2	EF			
3	EG	**CEG**		

Figure 14

In M_A for example, line 5, column 1, ACE is *in italics* since this sequence is incompatible with lines 1 through 4, column 0 of M_A: each time object type E is encountered, so is B, which means that E cannot occur without B. In ACE, E occurs without B, hence the incompatibility.

set, such that L is composed of the list of regions associated with the root A of G_H.

The general algorithm proposed is composed of two phases: P1 and P2. In the first phase, L is computed. In the second phase, using L as a basis, the single inheritance graph(s) equivalent

M_A	0	1	2	3	4	5
0	N1 to N0	**A**	**AB**	*AC*	**AD**	*ABC*
***********						
1	B					
2	BE			**ABCE**		
3	BEF			**ABCEF**		
4	BEG			**ABCEG**		
***********						
5	CE					
6	CEF					
7	CEG					

↑

M_B	0	1	2
0	N2 to N1	**B**	**BE**
***********			
1	E		
2	EF	**BEF**	
3	EG	**BEG**	

↑

M_C	0	1	2	3
0	N2 to N1	**CE**	*CF*	**CEF**
***********				
1	E			
2	EF			
3	EG	**CEG**		

Figure 15

Rule 2': this is rule 2 applied to line 0 of a given matrix. The application of rule 2' yields those presented in Figure 15.

In line 0 of M_C for example, CF is *in italics* since this sequence is incompatible with lines 1 through 3, column 0 of M_C: each time object type F occurs, object type E occurs along with it, which means that F cannot occur without E. In CF, F occurs without E, hence the incompatibility.

The final result of this process is L, the list of regions A, AB, AB, ABCE, ABCEF and ABCEG of matrix M_A. This is the same list as in the model shown in Figure 9, but obtained in a different way.

4. Formal Approach

Let G_H=(N, H, A) be a multiple inheritance graph. Let L be the subset of the set (denoted P*(N)) of the parts of N not including the empty

to the original multiple inheritance graph is (are) computed. N is broken down into a number of levels Ni with i=max to 0.

4.1 Phase 1 (P1)

Let $N_i^p = \{n_{ij} \in N_i / \Gamma_H^{-1}(n_{ij}) \neq \phi\}$

with $n'_{ij} \in \Gamma_H^{-1}(n_{ij})\ if \lfloor n'_{ij} n_{ij} \rfloor \in H$ and i=max−1 to 0. Therefore, $N_0^p = N_0 = \{A\}$ if H≠∅.

Let L_{ij}^{be} be the subset of the set (denoted $P^*(\Gamma_H^{-1}(n_{ij}))$) of the parts of $\Gamma_H^{-1}(n_{ij}) \cup (n_{ij})$ without the empty set, such that L_{ij}^{be} represents the list of regions associated with n_{ij}, <u>after</u> the constraints of exclusion and of totality have been satisfied (Figure 8) and <u>before</u> the possible contradictions which may appear when passing from one level to the next have been taken into consideration (Figure 9).

Let L_{ij}^{af} be the subset of the set (denoted $P^*(\Gamma_H^{-1}(n_{ij}))$) of the parts of $\Gamma_H^{-1}(n_{ij}) \cup (n_{ij})$ without the empty set, such that L_{ij}^{af} represents the list of regions associated with n_{ij}, after having taken into consideration the possible contradictions which may appear when passing from one level to the next. Therefore, L_{01}^{af} =L. Here is P1:

```
For i<—max-1 ; 0 ; step -1
    ∀ n ∈ Ni compute N_i^p
    For j<—1 ; |N_i^p| ; step 1
        compute L_ij^be
        L_ij^af<—L_ij^be
        For k<—1 ; |L_ij^be| ; step 1
            let l_ijk^be ∈ L_ij^be
            For p<—1 ; |Γ_H^-1(n_ij)| ; step 1
                let n_-p ∈ Γ_H^-1(n_ij)
                let L_-p^af
                For q<—1 ; |L_-p^af| ; step 1
                    let l_-pq^af ∈ L_-p^af
                    theoretical region<—l_-pq^af ∪ l_ijk^be
                    If (Rule 1)
                    theoretical region ⊃ x such that
                    x ∈ P*(Γ_H^-1(n_ij)) and x ∉ L_ij^be
                    Then L_ij^af<—L_ij^af ∪ {theoretical region}
                    End if
                End for
            End for
        End for
        BAD_ij^af<—∅
        For k<—1 ; |L_ij^af| ; step 1
            let l_ijk^af ∈ L_ij^af
            For p<—1 ; |Γ_H^-1(n_ij)| ; step 1
                let L_-p^af
                If (Rules 2 and 2')
                L_-p^af refuses: ³ l_ijk^af
                Then BAD_ij^af<—BAD_ij^af ∪ {l_ijk^af}
                End if
            End for
        End for
        L_ij^af<—L_ij^af - BAD_ij^af
    End for
End for
```

4.2 Phase 2 (P2)

M shall be the number of single inheritance graphs which are equivalent to the original multiple inheritance graph. $G'_{H'_m} = (N'_m, H'_m, A)$ shall represent a single inheritance graph equivalent to the original multiple inheritance graph with m=1 to M.

N_i^c shall be the set of candidate object types for the ordering at iteration i. The computation of N_i^c is immediate as long as the candidate object types have not been chosen yet and belong to the largest number of elements of L. Note that $|L|<2^{|N|}$. N_i^f shall be the set of object types chosen at iteration i with N_0^f =∅.

Principle of independence at iteration i: $n_j \in N_i^c$, $n_k \in N_i^c$, ∀ l ∈ L, n_j is independent of n_k if and only if $n_j \in l$ then $n_k \notin l$.

Principle of dependence at iteration i: $n_j \in N_i^c$, $n_k \in N_i^c$, let n_j D_i n_k (n_j is dependent on n_k according to D_i) if and only if ∃ l ∈ L such that $n_j \in l$ and $n_k \in l$.. D_i is a relationship of equivalence for N_i^c and breaks down N_i^c into equivalence classes called classes of dependence at iteration i. In addition, let ∀ $n_j \in N_i^c$ with j=1 to $|N_i^c|$, $D_{ij}=\{n_k \in N_i^c \ / \ n_j \ D_i \ n_k\}$. Let d_i be the number of classes of dependence at iteration i; in other words, the number of sets D_{ij} mutually different. Here is P2:

```
M<—1
i<—1
While N_i-1^f≠N
    ∀ n ∈ N-N_i-1^f compute N_i^c
    For j<—1 ; d_i ; step 1
        For k<—1 ; |D_ij| ; step 1
        n_k ∈ D_ij:    L_ijk={l ∈ L / n_k ∈ l} and
                       F_ijk={f_a ∈ N_i-1^f, f_b ∈ N_i-1^f / (∃ l_a ∈ L_ijk / f_a ∈ l_a and f_b ∈ l_a)
                       and (∃ l_b ∈ L_ijk / f_a ∈ l_b and f_b ∉ l_b)}
            For m<—1 ; M ; step 1
                For q<—1 ; |F_ijk| ; step 1
                f_q ∈ F_ijk:    P_m(n_k)=n_k.P_m(f_q) and
                                N'_m<—N'_m ∪ (∪{P_m(n_k)}) and
                                H'_m<—H'_m ∪ (∪{(P_m(n_k)P_m(f_q))})
                End for
            End for
            N_i^f<—N_i-1^f ∪ {n_k}
        End for
        M<—M*|D_ij|
    End for
    i<—i+1
End while (i≤|L|)
```

The introduction of the relationship Pm, which associates a set of substitution object types for each equivalent single inheritance graph and for each object type of N, can be illustrated by the following example:

Let L be the list of regions computed after phase 1: A, AB, AC, AD, ABC, ABD, ACD and ABE. Note that N_3^c={C, D}, D_{31}={C, D}, L_{311}={AC, ABC, ACD}, F_{311}={A, B}, L_{312}={AD, ABD, ACD} and F_{312}={A, B}.

It is thus substituted first at C, C1 and C2 such that $P_m(C)$ =C1 and $P_m(C)$ =C2 so that $N'_m \leftarrow N'_m \cup \{C1, C2\}$ and $H'_m \leftarrow H'_m \cup \{[C1A],[C2B]\}$.

It is then substituted at D, D1, D2, D3 and D4 such that $P_m(D)$=D1, $P_m(D)$=D2, $P_m(D)$=D3, $P_m(D)$=D4 so that $N'_m \leftarrow N'_m \cup \{D1, D2, D3, D4\}$ and $H'_m \leftarrow H'_m \cup \{[D1A],[D2B],[D3C1],[D4C2]\}$

5. Advantages and Disadvantages of the Proposed Technique

5.1 Disadvantages

There are two important problems posed by the proposed technique and its associated algorithm.

The first is the problem of generating a large number of single inheritance graphs equivalent to the original multiple inheritance graph. The root of this problem is the choice between several candidate object types in the process of ordering in phase 2 at iteration i. In the example at paragraph 3.2, the choice between object types C and E at iteration 3 poses precisely this problem. There are two possible solutions, not developed here. The first is to resolve the problem interactively by questioning the software engineer so as to be able to choose one object type over another (this is the choice made in [Barbier and Reich, 97]). The second solution is to compute, *a priori* if possible, the number of equivalent single inheritance graphs. If this is a large number, inform the software engineer on the intrinsic complexity of the original multiple inheritance graph.

The second problem is the increase in the number of object types of the original multiple inheritance graph. At the end of phase 2, a family of paths defines an equivalent single inheritance graph. During phase 2, if there exists q paths going from the object type chosen for ordering through to the root of the graph, it is necessary to introduce at least q object types in an equivalent single inheritance graph in order to replace the chosen object type of the original multiple inheritance graph. This problem can be illustrated as in Figure 16.

In the example shown in Figure 16, at iteration 3 of phase 2, there are the path from C to A which is ([CA]) and the path from C to A which

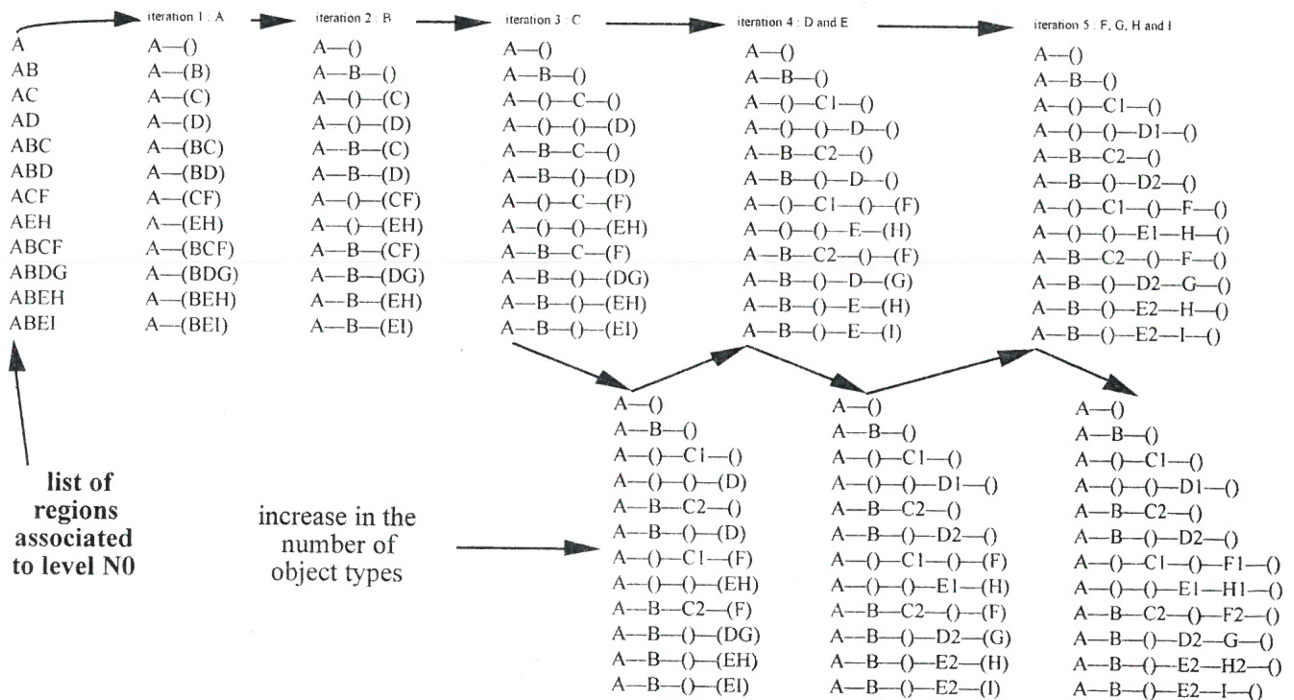


Figure 16

is ([CB], [BA]). The object types C1 and C2 are then introduced such that C1 → A and C2 → B. The final result of this process is the following single inheritance graph:

Thus, the model shown in Figure 17 remains easily usable, in spite of the increase in the number of object types. In general, the increase in the number of object types is acceptable

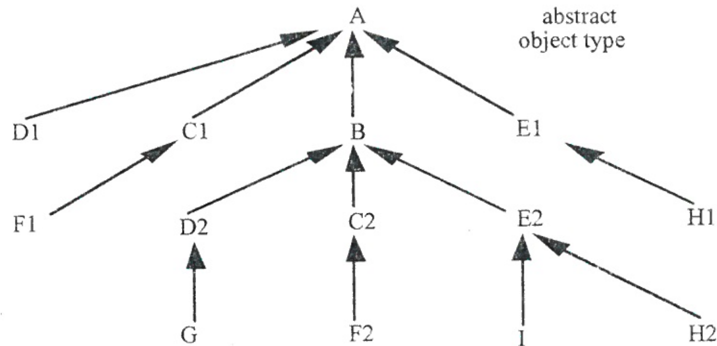


Figure 17

The original multiple inheritance graph was the following:

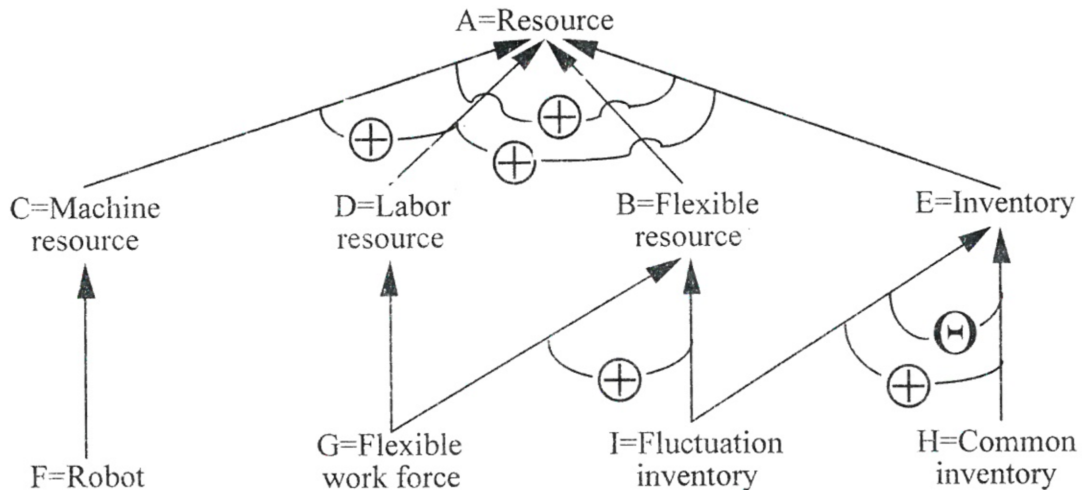


Figure 18

How could one interpret the model shown in Figure 17?

- there are two types of "Robot" (F1 and F2) because there are two types of "Machine resource" (C1 and C2). The first inherits directly from "Resource" (A), the second from "Resource" (A) via "Flexible resource" (B). Certain robots are flexible resources while others are not: is this a true statement? Is the model shown in Figure 18 valid?
- all "Flexible work force" (G) are a "Flexible resource" (B): is this a true statement? Is the model shown in Figure 18 valid?, etc.

within certain limits. In the above example, the increase of 9 to 14 object types is acceptable since the single inheritance graph can still be understood. However, there are certain examples where the increase is inacceptable, and these show that the original multiple inheritance graph was quite complex. One possible solution is to compute, once again *a priori* if possible, the increase in the number of object types. Here is an example of a multiple inheritance graph which generates too large a number of object types, when transformed into an equivalent single inheritance graph (an increase of 8 to 23 object types) as shown in Figure 19.

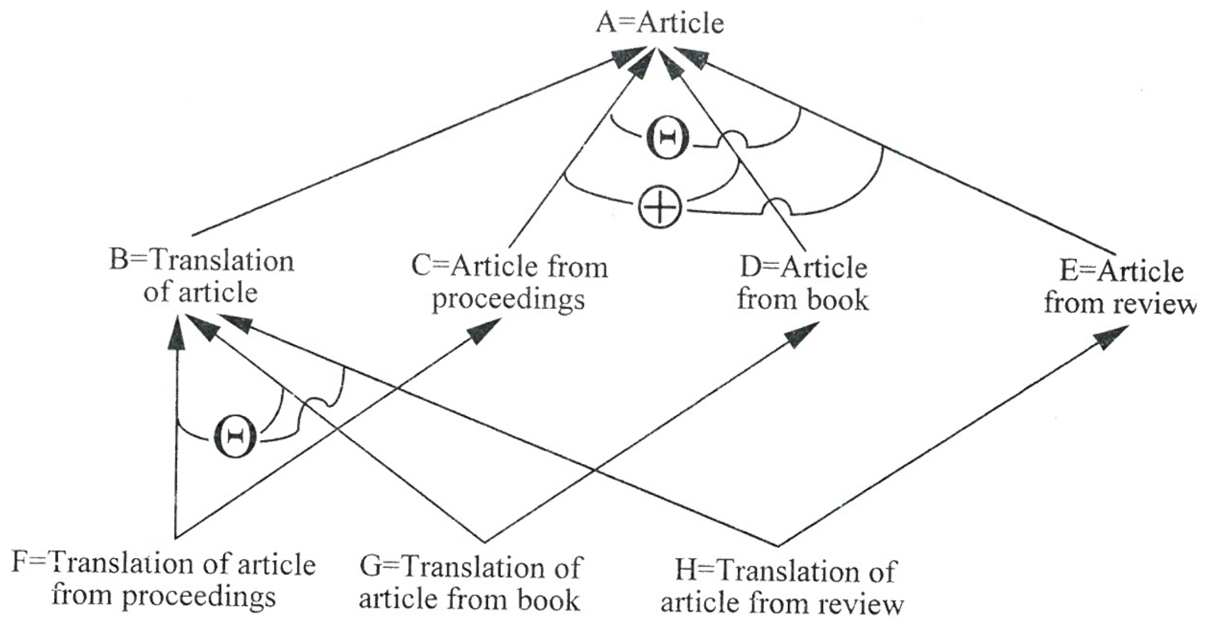


Figure 19

It is important to insist upon the fact that the technique here proposed is not one for implementation but for comprehension, and thus verification, of a multiple inheritance graph. For example, the model shown in Figure 17 is not meant to be implemented as it is. However, in the activity which follows analysis, the foremost considerations are those of implementation. This important aspect of the development process shall be discussed in the next Section.

5.2 Advantages

The main advantage of the proposed technique is the understanding of the model constructed. In the activity which follows analysis, the main question is to know, insofar as the model is valid, if the multiple inheritance graph or an equivalent single inheritance graph should be used in order to initiate the design of the software.

If the programming language does not employ multiple inheritance, there is no choice. Here are two pertinent examples:

Example 1

In the OMT method, multiple inheritance is advocated during the analysis activity [Rumbaugh et al, 1991, p.65]. There a technique of the implementation of OMT models with the help of relational databases is proposed [Rumbaugh et al, 1991, p.373]. Unfortunately (or voluntarily?), the implementation of multiple inheritance graphs was not addressed, while that of single inheritance graphs was: how should one then handle multiple graphs?

How can one justify an object-oriented method as an improvement over a traditional method if it cannot respond to this important —even fundamental— question? This criticism of OMT could possibly be equally applicable to other methods.

Example 2

Concerning the code generation function of the tool presented in [Barbier and Reich, 1997], a technique of implementation of the OMT models with the help of Smalltalk-80 is proposed. Since Smalltalk-80 does not employ multiple inheritance, it was necessary to use the verification technique presented in this article in order to solve the problem.

In general, even if the development system proposes the use of multiple inheritance, it is still critical to address the possibility of conflicts. Take the Eiffel object-oriented programming language as an example: it employs multiple inheritance, but leaves the resolution of conflicts up to the initiative of the software engineer. Within the realm of industrial development which has come to mean

essentially the automatic construction of software through the use of a software engineering tool, the technique herein proposed is a sure guide to defining strategies of implementation.

6. Conclusion

Inheritance appears to be a major asset in the software industry, whether one's priorities be with respect to conceptualization or with respect to implementation. However, concerning conceptualization priorities, the haphazard manner in which certain object-oriented methods employ inheritance, and in particular multiple inheritance, is incompatible with the systematic verification of object-oriented models needed by software engineers when using object-oriented software engineering tools.

Solving the problem of object-oriented model verification especially stumbles over the absence of a commonly accepted rich inheritance semantics. In this sense, the choice made in this article can contribute to methods interoperability, but it introduces a non sufficiently rich inheritance semantics. OMG results on methods interoperability are not available yet, while this does not prevent methods diffusion, this decreases methods credibility because tools cannot be powerful. A recent French study about users of object-oriented methods and tools tends to prove that software quality, in particular reusability and maintainability, is practically difficult to improve with the help of object-oriented methods and/or tools.

A current trend is to create formal object-oriented methods with efficient associated tools. As for these methods, this tends to complicate the definition of methods interoperability. As for these tools, this tends to strongly couple together application developments with a given tool, and consequently with a given method. Whereas a method must basically ensure programming languages-independent software developments, such an approach generates other problems because it becomes difficult to switch easily from one method notation to another.

ANNEX

```
"instances of class 'Regions'
embody areas depicted in Figure
8. This class implements the
'refuses:' method"

refuses: aRegion
  | refuses
aSubSetOfEachRegionsAfter
aDictionary |
  refuses := false.
  aRegion
    detect:
      [:eachRegionItem |
      aSubSetOfEachRegionsAfter
:= Set new.
      self do:
[:eachRegionsAfter1 |
(eachRegionsAfter1 includes:
eachRegionItem)
          ifTrue:
[aSubSetOfEachRegionsAfter add:
eachRegionsAfter1]].
      aDictionary := Dictionary
new.
      aSubSetOfEachRegionsAfter
do: [:eachRegionsAfter2 |
eachRegionsAfter2 do:

  [:eachRegionsAfter2Item |

  eachRegionsAfter2Item =
eachRegionItem
            ifTrue:
[eachRegionsAfter2 size = 1
ifTrue: [aDictionary at:
eachRegionsAfter2Item put: 1]]
            ifFalse:
[(aDictionary includesKey:
eachRegionsAfter2Item)
                ifTrue:
[aDictionary at:
eachRegionsAfter2Item put:

  (aDictionary at:
eachRegionsAfter2Item) + 1]
                ifFalse:
[aDictionary at:
eachRegionsAfter2Item put:
1]]]]].
      aDictionary isEmpty
ifFalse: [aDictionary keys
detect: [:eachKey | aRegion
includes: eachKey]
          ifNone: [refuses :=
true]].
      refuses | aDictionary
isEmpty ifFalse: [aDictionary
keys detect: [:eachKey |
                    refuses
:= (aRegion includes: eachKey)
not and:

  [(aDictionary at: eachKey) =
aSubSetOfEachRegionsAfter size]]
```

```
            ifNone: []].
        refuses]
      ifNone: [].
   ^refuses! !
```

REFERENCES

AGOPIAN, L., **Specification of Specialization Constraints with Venn Diagrams and Textual Language**, INFORSID '92 Proceedings, Clermont-Ferrand, France, 1992, pp.265-284.

ANDRÉ, P., BARBIER, F. and ROYER, J.-C., **An Experimentation of Object-oriented Formal Development,** COMPUTER SCIENCE AND TECHNIQUE, HERMÈS ÉDITIONS, Vol. 14, 8, 1995.

ATZENI, P. and PARKER, D.S., **Formal Properties of Net-based Knowledge Representation Schemes,** DATA & KNOWLEDGE ENGINEERING, 3, ELSEVIER SCIENCE PUBLISHERS, North-Holland, 1988, pp.137-147.

BARBIER, F., **Object-oriented Analysis of Systems Through Their Dynamical Aspects**, JOURNAL OF OBJECT-ORIENTED PROGRAMMING, SIGS Publications, Vol. 5, 2, 1992.

BARBIER, F. and REICH, G.-P., **Intelligent Software Factory/Analysis and Design,** ISF/AD USER'S GUIDE, Version 1.0 - 2.6, 1993 - 1997.

BLUMOFE, R. and HECHT, A., **Executing Real-time Structured Analysis Specifications**, SOFTWARE ENGINEERING NOTES, Vol. 13, 3, ACM, 1988.

BOURDEAU, R. and CHENG, B., **A Formal Semantics of Object Model Diagrams**, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. 21, 10, 1995.

COAD, P. and YOURDON, E., **OOA — Object-Oriented Analysis,** YOURDON PRESS, 2nd Edition, 1991.

COAD, P. and YOURDON, E., **OOD — Object-Oriented Design,** YOURDON PRESS, 1991.

COOK, S. and DANIELS, J., **Designing Object Systems, Object-oriented Modelling with Syntropy**, PRENTICE HALL, the Object-oriented Series, 1994.

DANFORTH, S. and TOMLINSON, C., **Type Theories and Object-oriented Programming**, ACM COMPUTING SURVEYS, Vol. 20, 1, 1988, pp.29-72.

DAVIS, A., **A Comparison of Techniques for the Specification of External System Behavior**, COMMUNICATIONS OF THE ACM, Vol. 31, 9, 1988.

DEMICHIEL, L.G., **Overview: the Common Lisp Object System**, LISP AND SYMBOLIC COMPUTATION, 1, 1988, pp.227-244.

DOD-STD-2167A, **Military Standard for Defense System Software Development**, ref. AMSC No. N4327, Department of Defense, Washington, D.C.,1988.

DORI, D. and TATCHER, E., **Embryonic Classes: Enabling Selective Multiple Inheritance**, JOURNAL OF OBJECT-ORIENTED PROGRAMMING, SIGS Publications, 1994, pp.36-40.

DUCOURNAU, R. and HABIB, M., **The Inheritance Multiplicity in Object-oriented Languages**, COMPUTER SCIENCE AND TECHNIQUE, HERMÈS ÉDITIONS, Vol. 8, 1, 1989.

DUCOURNAU, R. HABIB, M. , HUCHARD, M., MUGNIER, M.-L. and NAPOLI, A., **Latest Results Concerning Multiple Inheritance**, COMPUTER SCIENCE AND TECHNIQUE, HERMÈS ÉDITIONS , Vol. 14, 3, 1995.

GOLDBERG, A. and ROBSON, D., **Smalltalk-80, the Language and Its Implementation**, ADDISON-WESLEY, 1983.

HABRIAS, H., **The Binary Relational Model, IA Method (NIAM),** EYROLLES ÉDITIONS, 1988.

HAREL, D. and GERY, **Executable Object Modeling with Statecharts,** IEEE COMPUTER, 1997.

HUTT, A., **Object Analysis and Design, Description of Methods**, JOHN WILEY & SONS, OMG, 1994.

HUTT, A., **Object Analysis and Design, Comparison of Methods**, JOHN WILEY & SONS , OMG, 1994.

JACOBSON, I., CHRISTERSON, M., JONSSON, P. and ÖVERGAARD, G., **Object-oriented Software Engineering**, ADDISON-WESLEY, ACM Press, 1992.

LALONDE, W. and PUGH, J., **Subclassing ≠ Subtyping ≠ is-a**, JOURNAL OF OBJECT-ORIENTED PROGRAMMING, SIGS Publications, 1991, pp.57-62.

LIEBERHERR, K., **Adaptative Object-oriented Software, the Demeter Method with Propagation Patterns**, PWS , 1996.

MEYER, B., **Eiffel, the Language,** PRENTICE HALL, the Object-oriented Series, 1992.

MEYER, B., **An Object-oriented Environment, Principles and Application,** PRENTICE HALL, the Object-oriented Series, 1994.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F. and LORENSEN, W., **Object-oriented Modeling and Design**, PRENTICE HALL, 1991.

SELIC, B., GULLEKSON, G. and WARD, P., **Real-time Object-oriented Modeling**, JOHN WILEY & SONS, 1994.

SHLAER, S. and MELLOR, S.J., **Object-oriented Systems Analysis, Modeling the World in Data**, YOURDON PRESS, 1988.

SHLAER, S. and MELLOR, S.J., **Object Lifecycles, Modeling the World in States,** YOURDON PRESS, 1992.

TAIVALSAARI, A., **On the Notion of Inheritance**, ACM COMPUTING SURVEYS, Vol. 28, 3, 1996.