JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 1, January–February 2004

Checking Class Schema Usefulness

Jean-Claude Royer, OBASCO EMN/INRIA, cole des Mines de Nantes, France

In this paper we introduce a structural and object-oriented model. We present applications of this model to the checking of some ill-formed classes. We focus on static class diagrams mixing inheritance and composition relations. We consider an approach based on the notion of class usefulness, *i.e.* finitely generated and with at least one defined value. We show that this allows us to eliminate some wrong class designs or wrong schema designs. We present a general process to check this and a static algorithm which applies to the UML language.

1 INTRODUCTION

As quoted in [23, 22] we argue that in the future analysis will be model-driven, focused and partial. So early investment in modelling and analysis will be essential. On the one hand we believe that the value of abstract model will be greatly enhanced if a direct relationship with code can be established. On the other hand we expect the increasing importance of supporting sound and precise analysis.

One problem we study in this paper is finding a suitable approach to abstractly define class structure and means to check that such a structure defines at least one object. As an example, consider the design of lists: many users (even in some books [28, 12, 26]) consider a head (of type object) and a tail of type list. This is true as long as the concrete language provides a `void` value. However this is not satisfactory at the level of design or (formal) specification since we confuse an empty list with some non related values. Furthermore the use of `void` does not help in more complex situations.

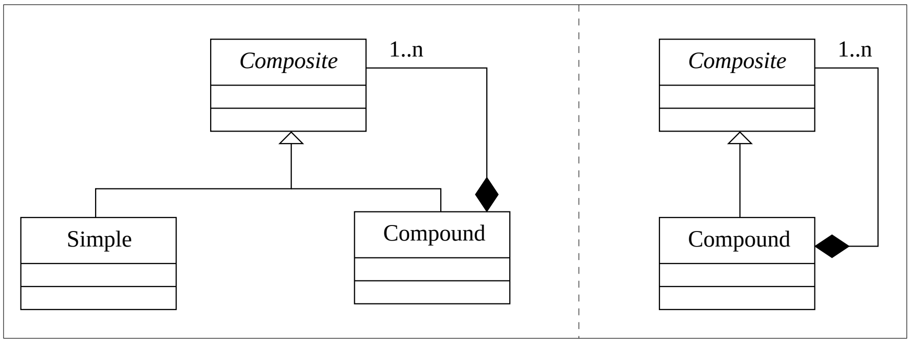


Figure 1: Two Variants of the Composite Pattern

Another example is the definition of designs mixing inheritance and composition.

Cite this article as follows: Jean-Claude Royer: *Checking Class Schema Usefulness*, in Journal of Object Technology, vol. 3, no. 1, January–February 2004, pages 157–176, http://www.jot.fm/issues/issue_2004_01/article4

For example, in the well-known composition pattern, in Figure 1, many users relax the `1..n` cardinality with *, but this is bad too since it defines at least a redundant model. More problematic is sometimes the existence of wrong definitions, which does not represent any generated and finite value. This is the case of the variant on the right in Figure 1 where the `Composite` class is declared abstract. In [13] a Z-based semantics for UML is presented with the support of a tool. A Section of this paper is related to Figure 1, however it is unclear what the authors consider as a correct design. They completely avoid the discussion about abstract classes, which may change definitively the correctness of a design.

While there is a great amount of literature and tutorials about inheritance *versus* composition (or aggregation), there is too few rules about the use of both the two concepts in a class diagram.

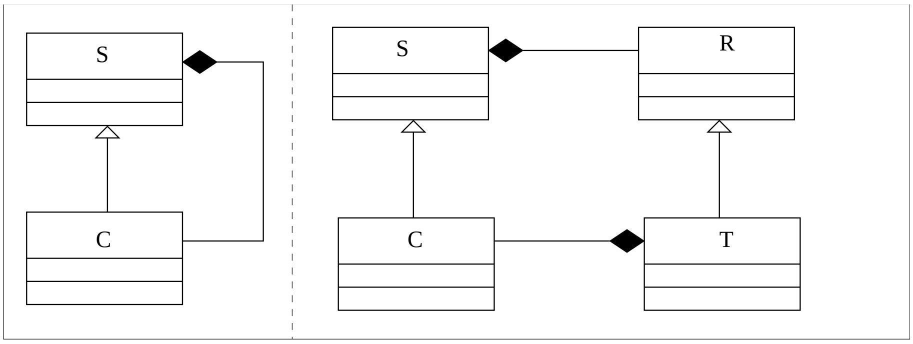


Figure 2: Some Wrong Diagrams

The left part of Figure 2 is surely not suitable from a software engineering perspective. But [13] considers it legal while [28] and [14] prohibit it. One of the oldest approach of this problem seems [28] which provides an object-oriented semantic model and gives a criterion. But the model considers only simple inheritance, concrete class, and an implicit void value; so the suggested criterion is incomplete. Their criterion says: *no directed cycle with an inheritance edge.* This criterion is false with the left picture but true with the right one. However these two pictures address the same problem, assuming that compositions are inherited a composition cycle appears at the fringe of the graphs. [14] defines an object model with concrete class, multiple inheritance and composition. Several rules are done and the rule 8 is relevant to our discussion. The rule states that the inheritance and composition graph must be without cycles. This is a wider rule than the previous one since it covers multiple inheritance and it also implies that composition graph is cycle-free. But it fails to check the right diagram because the rule does not cope with inheritance of compositions.

The current paper presents a natural idea, common to Abstract Data Type (ADT), to check classes and class diagrams. It is based on an algebraic model which allows concrete and abstract classes, conditional attributes and multiple inheritance. ¿From this we study conditions and algorithms to avoid such problems in UML designs. The approach exposed here may be useful to check various situations, for instance, recursive data type equations, or the use of inner classes and inheritance as in Java.

The paper is organised as follows. Section 2 gives a presentation of our model and its translation into partial first-order structures. Then, in Section 3 we detail the problem of class usefulness. Section 4 discusses the problem of usefulness for schemas and an algorithm in the total case is presented. Section 5 shows the application of our approach to the checking of UML static class diagrams. Finally, we present some related work and a conclusion summarises this presentation.

2 AN ALGEBRAIC MODEL FOR CLASSES

We present in this section our structural view of classes by translating them into partial first-order structures (PFOS, see [9]). Our basic model comes from the study of various object-oriented languages and their use in formal development. This model, called the *formal class* model is presented in [1] and a formal and operational semantics is described in [30]. The operational semantics is based on conditional term rewriting in a graph of sets of rules. This is a structural object-oriented data model as the Demeter kernel model [23]. The present paper focuses on how classes abstractly define objects taking into account the declared attributes and the inheritance relationships. We do not need here axioms for methods and we do not discuss issues related to inheritance and behavioural compatibility.

Some Algebraic Hypotheses

Following a common idea [24], we consider that a class is a particular representation of a data type. We consider that a "formal" class is intermediate between ADTs and concrete object-oriented classes. For the semantic issue we consider a loose approach. A first hypothesis is based on generators, we consider a single generator for each class. This constraint allows us to get a formal model of class with a straight translation into object-oriented languages.

It exists several representations of the same abstract data type into classes. Examples are finite lists which may be implemented, at least, in two different ways: with a unique class or with a hierarchical schema of three classes [30]. Our model does provide the ability to abstractly represent various designs of the same data type. We also need a kind of non-strict constructor as the `don't care` values of [4]. Nevertheless our need is simpler since undefined values are necessary but they are never the result of a computation. Let us consider the simple example of finite lists specified in Figure 3. The precise understanding of the PFOS formalism is not necessary to understand our approach. Note that $\mathrel{-\!\!\circ\!\!\rightarrow}$ means partial function, $\overset{e}{=}$ is existential equality, $\times$ is product type and other symbols are usual boolean connectors. The `newFlat` constructor is strict (as required by the considered algebraic model) relatively to the D definedness predicate. However a term built with `newFlat` may represent a correct value even if one of its subterms is not a defined value. The difference between correct and wrong values is done by the use of the $\perp_{Flat}$ constant and an additional level of interpretation. The $\perp_{Flat}$ value is well-defined relatively to the term definedness but it is not a correct value from a user point of view. This specification denotes finite lists plus a special value, which is often noted in denotational semantics by $\perp$. The need of $\perp$ arises because we are interested in finitely generated terms and we have strict recursive generators. The previous specification, one may found it a bit exotic, is straightforward to implement in any object-oriented languages.

```
spec flat enrich natural, elem by
  sorts Flat⊥
  opns
    ⊥_Flat :   ⟶  Flat⊥
  popns
    newFlat : Boolean × Elem⊥ × Flat⊥ -o-> Flat⊥
    head : Flat⊥ -o-> Elem⊥
    tail : Flat⊥ -o-> Flat⊥
  preds
    isEmpty : Flat⊥
  vars e : Elem⊥; l : Flat⊥
  axioms
    D(newFlat(e, h, t)) <=> (e ∧ (h =^e ⊥_Elem) ∧ (t =^e ⊥_Flat)) ∨
                            (¬e ∧ ¬(h =^e ⊥_Elem) ∧ ¬(t =^e ⊥_Flat))
    D(head(newFlat(e, h, t))) <=> ¬e ∧ ¬(h =^e ⊥_Elem) ∧ ¬(t =^e ⊥_Flat)
    D(tail(newFlat(e, h, t))) <=> ¬e ∧ ¬(h =^e ⊥_Elem) ∧ ¬(t =^e ⊥_Flat)
    D(newFlat(e, h, t)) => isEmpty(newFlat(e, h, t)) =^e e
    D(head(newFlat(e, h, t))) => head(newFlat(e, h, t)) =^e h
    D(tail(newFlat(e, h, t))) => tail(newFlat(e, h, t)) =^e t
endspec
```

Figure 3: A Partial Structure for Lists

A type is *strict* if it exists at least one finitely generated value of its sort, then the type has at least one non empty reachable algebra as model. In our context the definition is more complex due to the presence of $\perp$ values, it is called *usefulness*. A sufficient condition, classic in ADT is to require a *sensible* signature for every types, *i.e.* there is always one constant in each sort. This condition is generally too constraining in our context. We study this point in Section 3 and its checking is the basic idea of the consistency checking developed in this paper.

The Algebraic Presentation of a Formal Class

We consider a particular class of specification called *projective specifications*. They are built from a unique generator with a set of *field selectors*. These field selectors correspond to an abstraction of the class attributes but with the additional feature that they may be partial. This increases the expressive power of classes but also makes more precise the object description and hence avoid some ill-formed situations. As an example, in Figure 4, `tail` is partial because we have the precondition `requires: empty?(Self) = false`. If `tail` had been total then it would not have represented finite lists anymore.

`Flat`
inherit from `OBJECT`

aspect:

field selectors
`empty?` : `Flat` ⟶ `Boolean`
`head` : `Flat` -o-> `Nat`
requires: `empty?(Self) = false`
`tail` : `Flat` -o-> `Flat`
requires: `empty?(Self) = false`

Figure 4: A Flat Formal Class for Finite Lists

The Figure 4 represents the formal class description equivalent to the PFOS in Figure 3. We have three abstract attributes `empty?`, `head` and `tail` which are seen as operations (possibly partial) on the data type. This description can be directly implemented in any object-oriented languages without the risk of wrong and infinite definition. This is why we pay attention to the precondition of the field selectors. Such a class comes directly from the specification in Figure 3, we explain in the rest of this Section the translation principles. The general prototype for such a class is described in Figure 5, as previously noted it only represents the structural part (however $prec_i$ and $INVT_C$ are behavioural descriptions). This may be seen as a modern record with variant of Pascal [33].

`C`
inherit from `S1, ..., Sn`

aspect:

field selectors
$\texttt{sel}_1$: `C` -o-> $\texttt{T}_1$
requires: $\texttt{prec}_1$
...
$\texttt{sel}_n$: `C` -o-> $\texttt{T}_n$
requires: $\texttt{prec}_n$

constraint:
$\texttt{INVT}_C$

Figure 5: Prototype of the C Formal Class

The field selectors, $\texttt{sel}_1$, ..., $\texttt{sel}_n$, are observers which characterise the structure of the instances of a class. The *structuring types* of a projective specification are the types of the field selectors $\texttt{T}_1$, ..., $\texttt{T}_n$, they also appear as arguments of the generator. We note *dps* the graph of the relation C *has* T_i *as structuring type.* Difficulties may arise when this graph has directed cycles or circuits, in the above example we have `Flat` *dps* `Flat`, it will be called a *circular dps*. The type C may have an $INVT_C$ constraint (invariant as in Eiffel [24]) and the selectors may be partial with preconditions $prec_i$. The invariant is viewed as a common condition (preferably maximal) to the selector preconditions. In the rest of this paper, we will omit this invariant. This prototype has a straight correspondence with classes and it may be directly implemented in various languages. We have studied its translation into several languages: Eiffel, Java, Smalltalk, and C++.

Preconditions and invariant are boolean algebraic expressions built over the `self` variable which represents the object receiver. We assume that an object (or instance of a class) is characterised by its instance values, *i.e.* a *tuple of values* $v = (v_i : T_i)_{1 \leq i \leq n}$. To each instance we associate its *characteristic vector* $u = (u_i : Boolean)_{1 \leq i \leq n}$ which collects the field selector precondition values for v. Defined expressions over the `self` variable are equivalent to expressions built over the tuple of values $(v_i : T_i)_{1 \leq i \leq n}$. We call *characteristic formula* F_C the logic formula built over u and v and which checks if v is a tuple of values with vector u associated to class C.

The definition of the characteristic formula is:

$$F_C(u_1, ..., u_n, v_1, ..., v_n) \mathrel{\widehat{=}} \bigwedge_{i=1}^{n} (u_i = prec_i(v_1, ..., v_n)) \wedge (u_i = \neg(v_i = \perp_{T_i})))$$

It defines the set of the useful instances of a class. The first part copes with the selector precondition and the second part deals with the definition of the value. Note: we consider that every selector is partial and needs the use of $\perp$. In case of total selectors, the structuring type does not require a $\perp$ value, the precondition is true, and the second clause $(u_i = \neg(v_i = \perp_{T_i}))$ is not needed.

Translating Formal Classes into PFOS

¿From a formal class it is not too difficult to produce the corresponding projective specification. The transformation of the prototype in Figure 5 produces the algebraic specification in Figure 6. Computation of the signature is quite simple and axioms for selectors are straight. The result must verify some syntactic constraints, we do not present them here. The major problem is to explicit the definedness of terms, *i.e.* the Def_C, Def_1, ..., and Def_n expressions.

The characteristic formula allows us to compute the definedness in the following way: The definedness for a generator call is given by a kind of "Shannon formula":

$$Def_C(v_1, ..., v_n) \mathrel{\widehat{=}} \bigvee_{\substack{(u_i)_{1 \leq i \leq n} \in Boolean^n \\ \wedge \neg(\forall\ 1 \leq i \leq n\ u_i \neq false)}} F_C(u_1, ..., u_n, v_1, ..., v_n)$$

spec tad **enrich** $spec_1, ..., spec_m$ **by**
  sorts C
  opns
    $\perp_C$: $\longrightarrow$ C
  popns
    $newC : T_1, ..., T_n \overset{o}{\longrightarrow}$ C
    sel_1 : C $\overset{o}{\longrightarrow}$ T_1
      . . .
    sel_n : C $\overset{o}{\longrightarrow}$ T_n
  vars $X_1 : T_1$; . . . ; $X_n : T_n$
  axioms
    $D(newC(X_1, ..., X_n)) <=> Def_C(X_1, ..., X_n)$
    $D(sel_1(newC(X_1, ..., X_n))) <=> Def_1(X_1, ..., X_n)$
    . . .
    $D(sel_n(newC(X_1, ..., X_n))) <=> Def_n(X_1, ..., X_n)$
    $Def_1(X_1, ..., X_n) => sel_1(newC(X_1, ..., X_n)) \overset{e}{=} X_1$
      . . .
    $Def_n(X_1, ..., X_n) => sel_n(newC(X_1, ..., X_n)) \overset{e}{=} X_n$
endspec

Figure 6: The Resulting Transformation of the C Prototype

The definedness for selectors is similar:

$$Def_j(v_1, ..., v_n) \mathrel{\widehat{=}} \bigvee_{\substack{(u_i)_{1 \leq i \leq n} \in Boolean^n \\ \wedge\ \neg(\forall\ 1 \leq i \leq n\ u_i \neq false) \\ \wedge\ u_j = true}} F_C(u_1, ..., u_n, v_1, ..., v_n)$$

Below is the definedness for class `Flat` and its processing with the help of the Larch Prover tool [16]. The formula may seem complex, but it contains many trivial simplifications and the Larch Prover tool simplifies it with no further axioms.

```
% Line beginning with % is a Larch Prover comment
% Larch Prover definition of F for class Flat
assert F(u1, u2, u3, v1, v2, v3) = ((u1) /\
                                    (u2=(~v1)) /\ (u2 = ~(v2=botNat)) /\
                                    (u3=(~v1)) /\ (u3 = ~(v3=botFlat)))

% Larch Prover definition  for DefFlat
assert DefFlat(v1,v2,v3) = (F(true,true,true,v1,v2,v3) \/
                            F(true,true,false,v1,v2,v3) \/
                            F(true,false,true,v1,v2,v3) \/
                            F(true,false,false,v1,v2,v3) \/
                            F(false,true,true,v1,v2,v3) \/
                            F(false,true,false,v1,v2,v3) \/
                            F(false,false,true,v1,v2,v3))

% automatic reduction by the Larch Prover to
```

$$\begin{aligned} Def_{Flat}(v1, v2, v3) = (&\neg(v3 = \perp_{Flat}) \wedge \ \neg(v2 = \perp_{Nat}) \wedge \ \neg v1) \\ &\vee\ (v3 = \perp_{Flat}) \wedge \ (v2 = \perp_{Nat}) \wedge \ v1)) \end{aligned} \tag{1}$$

3 CLASS USEFULNESS AND ITS CHECKING

An edge of the *dps* relation is said *total* if its precondition is true else it is called *partial*. These definitions are extended to sets of edges.

Definition 3.1 *A useful instance is different from $\perp$ and its tuple of values is either empty or it has at least one useful value and the associated precondition is true.*

Definition 3.2 *The C class is useful if and only if it defines at least one useful instance.*

We naturally assume that predefined types are useful (in fact they are even strict). The characteristic formula defines the set of useful instances of a class. If $n = 0$ then there is only a unique instance for the class. If $n > 0$ then F_C must have a solution tuple.

Lemma 3.3 *C is useful if and only if either $n = 0$ or else Def_C is satisfied.*

This is for example the case of the `Flat` class, which has two different characteristic vectors: $\{(true, false, false), (true, true, true)\}$. This leads to the two well-known kinds of useful instances: empty lists with tuple $(true, \perp_{Nat}, \perp_{Flat})$ and non empty lists with tuple $(false, v_2 : Nat, v_3 : Flat)$. It seems harder to find a general and necessary condition for class usefulness; in fact the use of preconditions prohibits to get a general and static criterion.

Lemma 3.4 *If C is useful then its dps graph has no total circuit.*

We have a well-founded subterm ordering because useful instances are finitely generated and thus the *dps* relation has no total circuit.

To check usefulness is more complex when *dps* is circular and partial. Since we allow preconditions, the composition relation may be circular but it cannot be anything[1]. A common case is when the C class has only self-circular dependency. *Omitting the self-circular dps* means to take $v_i = \perp_C$ and $u_i = false$ for all $v_i : C$ in the characteristic formula.

Lemma 3.5 *Let be a C class that has only self-circular dps, C is a useful class if and only if it exists a useful value omitting the self-circular dps.*

Since we have a well-founded subterm ordering, it exists a least useful instance which has $\perp_C$ values for self-circular attributes. The opposite way is trivial. This lemma applies for example in the case of the `Flat` class. The formula 1 is reduced by taking $v_3 = \perp_{Flat}$ and a useful instance is described by the tuple $(true, \perp_{Nat}, \perp_{Flat})$. Tools like theorem provers, logic programming languages, or constraint solvers may help to satisfy this formula. Below is an example processed with the Larch Prover tool.

[1] This remark is also valid in an imperative model whenever we distinguish object structures from object pointers.

```
% proof of the usefulness of Flat[Nat]
prove \E v1 \E v2 DefFlat(v1, v2, botFlat)
% automatic simplification done by Larch Prover
 Current subgoal: \E v1 v1
 % satisfaction is immediate but manual
 resume by specialisation u1 to true

   Conjecture user.4
   [] Proved by specialization.
```

Let be $\mathcal{T}$ the transformation where: The invariant is changed to $true$ and the partial edges are changed to $false$.

Lemma 3.6 *If C is a useful class then $\mathcal{T}(C)$ is a total and useful class.*

It can be proved from the definition of class usefulness.

Structural Inheritance

The notion of inheritance we consider here deals only with the structural aspects of classes. A previous work [29] defines inheritance on a set theoretical basis for classes which may have several structural descriptions. [30] introduces a rewriting approach to method inheritance. Our objective was to get inheritance more rigorous. We assume that name conflicts have been solved using renaming or overriding [24]. The inheritance definition is based on a matching of selector names between the field of the S superclass and the fields of the C subclass. It defines a coercion called the *structural projection*, which is a partial homomorphism between the Σ-structures associated to the classes. In [29] we proved a static and necessary condition for the existence of this structural projection.

Criterion 3.7 (Inheritance Criterion) $C\ ako\ S \implies \forall fsel_i^S : S \mathrel{-\!\!\circ\!\!\!\rightarrow} T_i,\ \exists fsel_i^C : C \mathrel{-\!\!\circ\!\!\!\rightarrow} R_i,\ (R_i = T_i) \vee (R_i\ ako\ T_i)$

This approach allows subtyping, but furthermore it implies that methods of the superclass are inherited in a safe way by the subclass.

With inheritance it is relevant to consider two distinct sets of instances: the set of effective instances (which are instantiated from the class) and the set of all the instances (which takes into account instance polymorphism), see [17, 29] for more details and examples of definitions. In the following Def_C is the characteristic function of the set of effective instances and H_C is the characteristic function of the set of all the instances. Let $\{S_j < C\}$ the set of the strict and direct subclasses of C. Instance polymorphism is defined as formula H_C, which denotes if a tuple of values is an instance of C or of one of its subclasses.

$$H_C = Def_C \bigvee_{\{S_j < C\}} H_{S_j}$$

Inheritance introduces instance polymorphism but also the notions of abstract class and concrete class. The definition for concrete class usefulness is now the following:

Definition 3.8 *Let be C a concrete class, it is useful if and only if H_C has a solution.*

We consider abstract classes since they play an important role in object hierarchies. Contrary to a concrete class, the characteristic formula of an abstract class would be $Def_C = false$. Nevertheless the case of abstract classes is more subtle because they are sometimes useful and sometimes not. For example, if an abstract class is at the fringe of the inheritance graph, we consider that such an abstract class must be useful in some way, *i.e.* Def_C must be satisfiable.

Definition 3.9 *Let be C an abstract class, it is useful if and only if Def_C has a solution and either C is a leaf of the inheritance graph or else $\bigvee_{\{S_j<C\}} H_{S_j}$ has a solution.*

The rationale for this choice is: To minimally subclass an abstract class is to add a concrete subclass with no further information and it must lead to a useful subclass. We must allow abstract class at the fringe of the inheritance graph because it is an important feature of the class library design. Indeed several variants are possible here, we think that our choices are sensible. Nevertheless other ways seem right, see [23] for a different choice, and the discussion related to Figure 9.

4 CHECKING SCHEMA USEFULNESS

A *schema of class* is a set of classes linked by inheritance and structural dependency relations. It is commonly accept as true that *ako* is not circular and we have already seen that *dps* is not total circular. We also assume that our inheritance definition is satisfied, hence the criterion 3.7 is true.

Useful Schema

Definition 4.1 (Useful Schema) *A class schema is useful if and only if all of its classes are useful.*

To check if a class schema is useful may become complex since we have mutually recursive equations and generally large size class diagrams. It is not prominent that

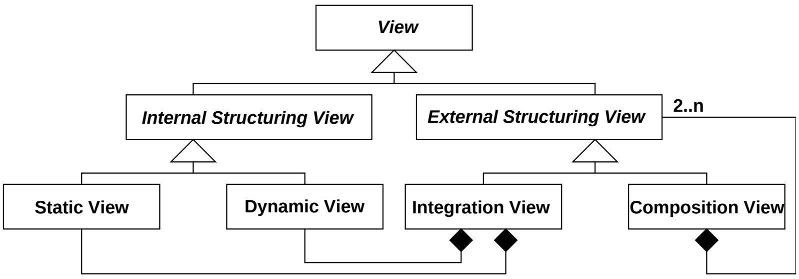


Figure 7: The View Diagram of Korrigan

the schema of Figure 7 is useful.

One approach, presented in [3], is to build, *a priori*, correct schemas from recursive data type equations. But this requires to formalise the data type with equations, and it does not generate the best object-oriented class design.

The verification principles are the following: We build a graph where the nodes are the equations to solve and where edges are the dependence between equations. For a class, we have two nodes Def_C and H_C. The edges starting from a given node are labelled with `and` or `or` to denote the kind of constraint among them. To build the graph we use the patterns in Figure 8. The process to verify the usefulness of

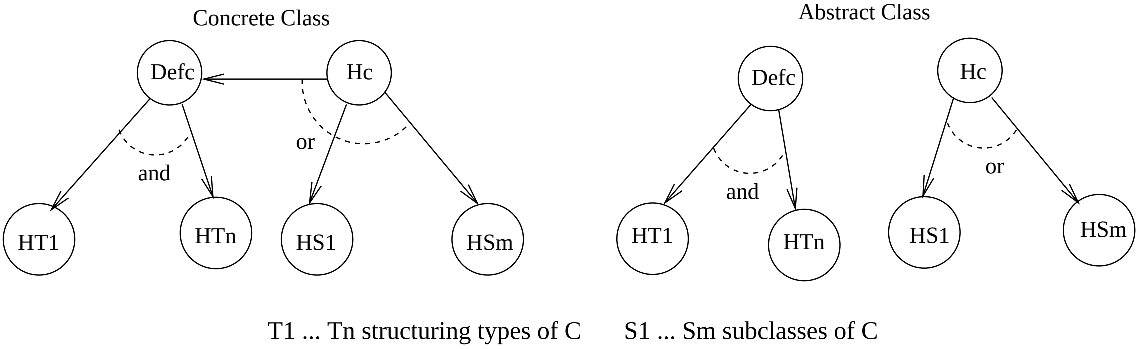


Figure 8: Construction of the Verification Graph

a class schema is based on putting marks on the nodes of the verification graph. To mark a node implies that the corresponding formula is satisfied by at least one useful value. At the beginning no node is marked and, if the process successfully terminated, at the end all nodes are marked.

Process 4.2
We iterate the following rules until all nodes are marked or no rule applies. These rules have as premise "if the node is not marked".

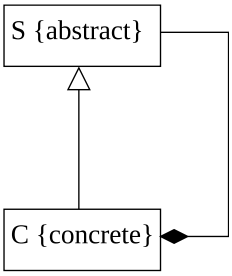
1. *If the node has no starting edges then it is marked.*
2. *If the node is linked with another node marked by a* `or` *then the former node is marked.*

3. *If the node has an* **`and`** *starting edge and all its neighbours are marked. We try to satisfy the* Def *formula as we have discussed in Section 3 and if this succeeds then the node is marked else the verification process fails.*

A class schema is useful if and only if its verification graph is solvable using the previous verification process. The general process is not decidable, but a limited static criterion is developed in Subsection 4.

The Smallest Wrong Example

This process succeeds in the example Figure 7, but it fails in the example Figure 9. The verification process shows a circuit in the satisfaction of the formulas.



	S {abstract}	**S {concrete}**
C {abstract}	wrong^a	useful^b
C {concrete}	wrong^c	useful^d

Figure 9: The Smallest Wrong Example and Its Variants

We have previously seen a necessary condition in the total case: The dependency graph is not circular. This example shows that wrong schemas, in the total case, comes from circuits in the $dps \cup ako^{-1}$ graph. But the criterion is not simple because we have abstract classes and changing the status of a class may transform a wrong schema into a useful one and *vice versa*. The right part of Figure 9 summarises the different cases for this example. The a, b, c, d cases illustrate different and common situations. Case a is wrong, if we subclass the `C` class with a concrete class it leads to a wrong schema. The b case is here considered as acceptable (many languages allow it), but we may add conditions to avoid it. Case c is typically a wrong example of schema and conversely case d is a classic and useful example.

A Static Criterion for the Total Case

A simple static criterion is possible in the total case by direct verification on the class schema.

Lemma 4.3 *A concrete class is useful if and only if it only and structurally depends from useful classes or it has at least one useful subclass.*

Lemma 4.4 *An abstract class is useful if and only if: It only and structurally depends from useful classes and, either it is a leaf of the inheritance graph or it has one useful subclass.*

These two lemmas comes from Definitions 3.8 and 3.9 and from the fact that in the total case solving Def_C reduces to the usefulness of the structuring types of C.

A term rewriting approach is possible to check if a given class schema is useful. First we define class schemas as values of a data type `System` and we formalise the previous lemmas as conditional term rewriting rules. This was done with the help of the Larch Prover tool, which proves the left-to-right rules terminating (under the predefined `noeq-dsmpos` ordering) and the system without critical pair. This automatically proves that the rewriting system is convergent: It computes a function. We have also implemented a simple version of the algorithm in Python and its complexity is the following. Let be n the number of class, $nAko$ the maximum number of subclasses, and $nDps$ the maximum number of structuring types; the algorithm loops n times and each time it checks all the classes. Each check costs no more than $nAko + nDps$. The overall complexity is less than $n^2 \times (nAko + nDps)$ and we think that some optimisations are still possible.

In the general (partial) case the previous algorithm gives a necessary condition to get a useful class schema. The graph may be transformed into a total schema by relaxing the invariants and the preconditions using the $\mathcal{T}$ transformation of Section 3.

Lemma 4.5 *If a partial class schema S is useful then the $\mathcal{T}(S)$ schema is total and useful.*

The sketch of the proof is the following. Since *ako* is a well-founded ordering, we have classes at the fringe and lemma 3.6 applies to them. If a class is a node in the inheritance graph of $\mathcal{T}(S)$, it is useful because its subclasses are useful.

5 APPLICATIONS TO THE UML LANGUAGE

We illustrate in this Section the concrete application of our work to the UML language. First of all, our approach is more general on several points: Typing, conditional attributes, and inheritance criterion. Second we are only interested in checking usefulness of classes, *i.e.* a limited kind of inconsistency checking: There is no class with an empty set of instance. In UML there are several diagrams, but we focus on static class diagrams. In this kind of diagrams, we consider classes (abstract and concrete), typed attributes, inheritance, aggregation, and composition relations. Interfaces may be viewed as particular abstract classes without attributes and with subtyping. General associations do not denote something about usefulness but they bring informations about the number of instances and other kinds of inconsistency checking are possible (see [15, 7, 32] for examples).

Another point has to be quoted here, [6] explains that the current official semantics of UML restricts the interpretation of classes to finite sets of objects. That is an obviously strict and weak semantics. The author proposes to use OCL constraints to avoid ill-formed schema. However its proposal does not have a simple link with object-oriented programming. Our theoretical approach shows that we may have a natural semantics for classes with an infinite set of objects in UML. Nevertheless we have to check some rules to avoid ill-formed schemas.

We assume that our inheritance criterion is true with the UML diagrams, a simple parsing over the diagram may ensure it. We do not interpret OCL, but for instance [18], shows the link between ADT or first-order logic and OCL. Thus a translation from OCL into our algebraic expressions is possible. Using this translation our general process applies in the case of UML. This general process is interesting but surely only useful for specialists, we need an automatic technique more adequate to UML designers. In the rest of this Section we present a limited but automatic way based on our criterion for the total case.

The *composition* relation (represented as a black diamond) in UML is not circular at the level of instances. This comes from the fact that the relation is transitive, antisymmetric and prohibit instance sharing (see [19] for a more detailed discussion). At the level of class, composition is called structural dependency in our model. A class composition (or equivalently the *dps* relation) is total if the minimal cardinalities are strictly greater than 0 (or the selectors have true preconditions). We consider that aggregation is a composition with object sharing. We restrict the static diagrams: There is no code description and the cardinalities are limited in the following way: For a composition or an aggregation, we only consider the navigation from the composite to the component and the cardinalities $\{0\}$ or $\{1\}$. This is done by taking the minimal cardinality, because we are interested in the existence of at least one instance. The algorithm of the total case 4 applies and as Lemma 4.5 states, this gives a necessary criterion for usefulness.

The schema of Figure 10 represents a preliminary UML design of a tool in the KORRIGAN environment [11]. It defines different kinds of state/transition diagrams: classic labelled state machines, symbolic machines with guards, with mail boxes, asynchronous machines and so on. It has neither inheritance nor composition cycles, criteria of [28, 14] do not apply since we have abstract classes. We check it with our algorithm and we found an error.

We have processed several middle case studies of the literature and we mainly found useful schemas (an example of a ill-formed schema may be found in [8]). The published diagrams have often a small size, they have been designed by specialists and many times they have been implemented and such errors have been fixed. As mentioned in the introduction, designers strongly relax the cardinalities by putting 0 or *. Designers often abuse of associations where composition or aggregation relations would be more adequate. These remarks explain that errors are not prominent in the diagrams from the literature. But as our examples illustrate it, such a tool is useful during the class design.

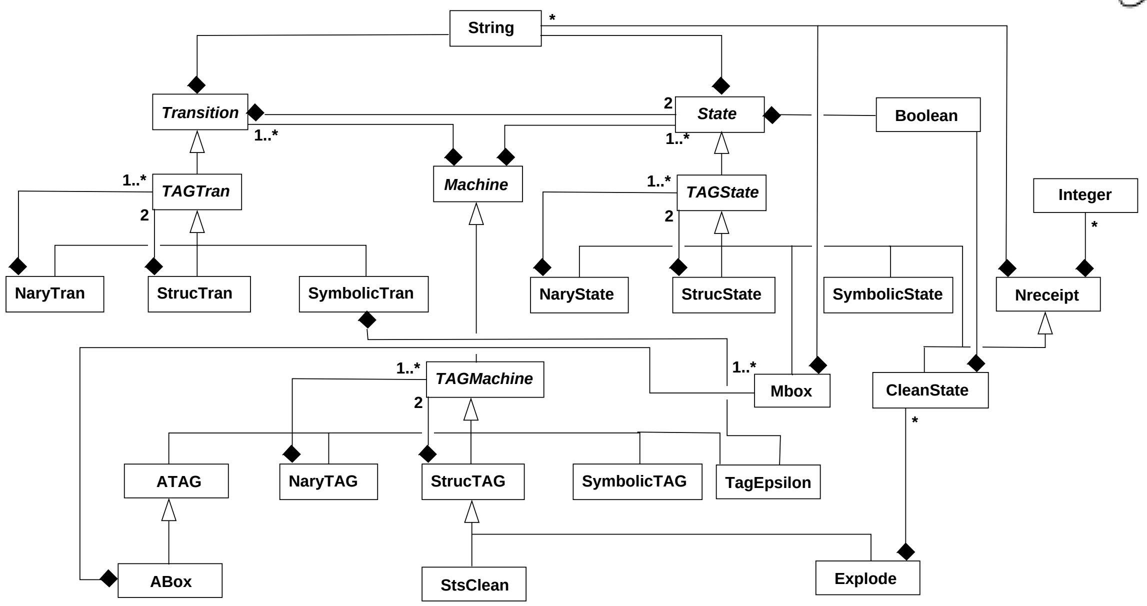


Figure 10: The TAG Schema

Related Work

[23] introduces an axiomatic object-oriented and structural data model called the Demeter kernel model. This model defines class dictionary graph which is a set of constructions and alternations. This is similar to our model but we have a more powerful structural model with partial informations. Another important difference is: in Demeter superclasses are always abstract classes and leaves of the inheritance graph are concrete classes. Our choices are less strict than those of Demeter and match some languages like Eiffel with no explicit interfaces. Demeter provides also some rules but nothing related to our class usefulness checking. The cycle-free alternation rule is simply: *no circuit in the inheritance graph.* Our total criterion applies to the class dictionaries of Demeter and gives the following simple criteria:

- *A concrete class is useful if and only if it structurally depends from useful classes*
- *An abstract class is useful if and only if it has a useful concrete subclass.*

One of the oldest reference which gives a rule about this problem seems [28]. It presents a model but without selector preconditions, without multiple inheritance, without abstract class and without care about $\perp$. The purpose of this book is to define a copy-and-paste semantics, and also to provide type-checking and inference. One aspect of this work is the well-formedness requirement which is linked to separate compilation. This work allows *dps* circuits ($has - a$ relation), but void values are implicit in each class, they are always useful. Our criterion is based on a more basic need than separate compilation: We want to be sure that each class may describe at least one finitely generated value. Our model is more general and we do not tackle separate compilation.

V. Engelson [14] presents a simple and more general rule in a context with concrete classes and multiple inheritance. This rule states *no circuit in the inheritance and composition graph*. It implies the previous one, the cycle-free alternation of [23] and the *dps* without circuit. However it does not cope with abstract class and does not explicit inheritance of compositions as illustrated with Figure 2.

There is now a real interest in checking consistency of UML specifications. We may point out three levels of checking for such a language: Purely syntactic checking, static semantic checking or verification of the dynamic semantics. UML is a complex language, the different approaches focus only on few aspects: Mainly on static diagrams, Statecharts, message sequence charts or activity diagrams. One may also note that consistency is *intra*-model or *inter*-models. Our work presents an automatic means to check a related consistency property inside static class diagrams. We avoid in the sequel to discuss several important work about checking behavioural specifications because it is out of the scope of this paper.

Many tools are able to type-check and to control syntax, but they seem really limited as we may read it in [25]. There are also tools for the static semantic level as [10], which checks the conformity of UML model against the OCL MOP description. [5] proposes to formalise the consistency conditions that must hold between model components. This is of course not sufficient because OCL and the meta-object description have not been proved consistent and they are not yet stable. There are now several work about checking inconsistencies in UML at the semantic level [21, 20, 27, 2, 31, 7]. Often they use a translation into labelled transition systems hence applying model-checking. Other approaches prefer to use a theorem prover approach for several reasons: This increases the expressive power since they are not limited to finite data types or state machines. For instance in [2], using our model and the Larch Prover tool, we have shown how to check general inconsistency problem in the UML static class diagrams. The main problem is that tools are only useful for proof specialists, the designers need automatic decision procedures. This one of the important benefit of model-checking.

Our current work is an example of an automatic decision procedure related to the general problem of consistency. Here this is not the general consistency problem of first-order logic but only a sub case. This is a similar situation as in [15] which studies cardinality checking using the idea that classes and associations must have at least one non-trivial model. [32] develops a related approach to check consistency between classes and sequence diagrams based on attribute typed graphs. They do multiplicity checking but the approach seems not very useful from a practical point of view since class cardinalities are often unknown.

6 CONCLUSION

In this paper, we present a formal and structural model of classes based on partial first-order structures. This model has a straight link to object-oriented programming and allows various descriptions (with or without inheritance) of the same data type. Our formal model has a uniform and powerful abstraction of instance variables, which permits precondition on the field selectors. This allows us to adapt the notion of strictness to our classes. Due to the presence of recursive and partial generators, the adequate concept is called class usefulness. We herein develop a general process to check class usefulness either at the level of class, *i.e.* with only composition, or at the level of class schema, *i.e.* taking into account inheritance. A static criterion is described and was implemented for the total case, *i.e.* whenever the preconditions are assumed to be true. We discuss the case of abstract classes which introduce the need for specific rules to check class usefulness. We illustrate this issue on several different examples common in pattern descriptions. We also show results which allow the application of this criterion to the general case, *i.e.* with partial preconditions for the field selectors. Lastly, we detail the use of these techniques to the case of some parts of the UML class diagrams. We have a static criterion, which is useful to check class usefulness with abstract and concrete classes, interfaces, inheritance, composition and aggregation relationships.

We have presented the basic principle of our consistency checking but two additional remarks may be done. It is possible to adapt the algorithm to propose categories, abstract or not, for classes to get the acceptable solutions. An efficient algorithm must be developed for that, since at first sight the complexity is $2^n \times n^2 \times (nAko + nDps)$. Our future work includes refining this study and proposing algorithms to check redundancy in class schemas and to suggest the best choice of cardinalities.

Acknowledgments

The author thanks Yann-Gal Guhuneuc and Pascal Poizat for their careful reading of previous versions of this paper, and their insightful comments.

REFERENCES

[1] Pascal André, Dan Chiorean, and Jean-Claude Royer. The Formal Class Model. In *Joint Modular Languages Conference, Modula, Oberon & friends*, ISBN 3-89559-220-X, pages 59–78, Ulm, Germany, 1994.

[2] Pascal André, Annya Romanczuk, Jean-Claude Royer, and Aline Vasconcelos. Checking the Consistency of UML Class Diagrams Using Larch Prover. In T. Clark, editor, *Proceedings of the third Rigorous Object-Oriented Methods Workshop*, BCS eWics, ISBN: 1-902505-38-7, 2000. http://www.ewic.org.uk/ewic/workshop/view.cfm/ROOM2000.

[3] Pascal André and Jean-Claude Royer. La modélisation des listes en programmation par objets. In Pierre Cointe, Christian Queinnec, and Bernard Serpette, editors, *Journées Francophones des Langages Applicatifs (JFLA'94)*, number 11 in Collection Didactique, ISBN : 2-7261-0824-5, pages 259–285, Noirmoutier, 1994. INRIA.

[4] Egidio Astesiano and Maura Cerioli. Non-strict don't care algebras and specifications. *Mathematical Structures in Computer Science*, 6(1):85–125, 1996.

[5] Hnatkowska B., Huzar Z., and Magott J. Consistency Checking in UML Models. In *4th International Conference on Information Systems Modelling (ISM '01)*, 2001. http://www.fit.vutbr.cz/events/ism/2001/.

[6] Thomas Baar. Metamodels without Metacircularities. *RSTI - L'objet, 4th Rigorous Object-Oriented Methods Workshop*, 9(4):95–114, 2003.

[7] Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Unifying Class-Based Representation Formalisms. *J. of Artificial Intelligence Research*, 11:199–240, 1999.

[8] Dulcinea Carvalho, Roy Campell, Geneva Bedford, and Dennis Mickunas. Definition of a User Environment in a Ubiquitous System. In D. C. Schmidt R. Meersman, Z. Tari and al., editors, *On The Move to Meaningful Internet Systems 2003: Coopis, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1151–1169. Springer Verlag, 2003.

[9] M. Cerioli, T. Mossakowski, and H. Reichel. From Total Equational to Partial Conditional. In H.J. Kreowski, B. Krieg-Brueckner, and E. Astesiano, editors, *Algebraic Foundation of Information Systems Specification*, chapter 3, pages 31–104. Springer Verlag, 1999.

[10] Dan Chiorean, Adrian Crcu, Mihai Pasca, Cristian Botiza, Horia Chiorean, Sorin Moldovan, and Ilinca Ciupa. A Framework for Checking UML Models - The UBB OCL Evaluator . Poster 6 at the 16th European Conference on Object-Oriented Programming, 2002.

[11] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. The Korrigan Environment. *Journal of Universal Computer Science*, 7(1):19–36, 2001. Special issue: Tools for System Design and Verification, ISSN: 0948-6968.

[12] Roland Ducournau. Spcialisation et sous-typage : thme et variations. *RSTI Techniques et Sciences Informatique*, 21(10):1305–1442, 2002.

[13] Sophie Dupuy, Yves Ledru, and Monique Chabre-Peccoud. Vers une intgration utile de notations semi-formelles et formelles : une exprience en UML et Z. *L'objet*, 6(9–32):21–47, 2000. ISSN 1262-1137.

[14] Vadim Engelson. ObjectMath Inheritance and Composition Diagram Editor. *Linkping Electronic Articles in Computer and Information Science*, 5(6), 2000. http://www.ep.liu.se/ea/cis/2000/06/.

[15] Pascal Fradet, Daniel Le Métayer, and Michaël Périn. Consistency Checking for Multiple View Software Architectures. In Oscar Nierstrasz and Michel Lemoine, editors, *Proceedings of the 7th European Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engeneering*, volume 24.6 of *Software Engineering Notes (SEN)*, pages 410–428, N. Y., September 6–10 1999. ACM Press.

[16] Stephan Garland and John Guttag. An overview of LP, the Larch Prover. In *Proc. of the 3rd International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[17] Giorgio Ghelli and R. Orsini. Types and Subtypes as Partial Equivalence Relation. In *Workshop on inheritance hierarchies in knowledge representation and programming languages*, pages 129–140, February 1989.

[18] A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proceedings of Asia Pacific Conference in Software Engineering*. IEEE Press, January 1998.

[19] Brian Henderson-Sellers and Franck Barbier. Are UML's Aggregation Kinds Meaningful. *L'objet*, 5(3-4):21–47, March 2000. ISSN 1262-1137.

[20] W.M. Ho, F. Pennaneac'h, and N. Plouzeau. UMLaut: A Framework for Weaving UML-based Aspect-oriented Designs. In *In Technology of object-oriented languages and systems (TOOLS Europe)*, pages 324–334, 2000.

[21] Daniel Jackson. A Comparison of Object Modelling Notations: Alloy, UML and Z. http://sdg.lcs.mit.edu/ dnj/publications.html.

[22] Daniel Jackson and Martin C. Rinard. Software analysis: A roadmap. In *Proceedings of the 22th International Conference on Software Engineering (ICSE-00)*, pages 133–146, NY, June 4–11 2000. ACM Press.

[23] Karl J. Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. From objects to classes: Algorithms for object-oriented design. *Journal of Software Engineering*, 6(4):205–228, July 1991.

[24] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Ed.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1997.

[25] Michael Moors. Consistency Checking. http://www.therationaledge.com/rosearchitect/mag/current/spring00/f6.html, apr 2000.

[26] Kasper Osterbye and J. Olsson. Scattered Associations in Object-Oriented Modeling. In *Proceedings of Nordic Workshop on Programming Environment Research*, June 1998. Bergen, Norway.

[27] Richard F. Paige, Jonathan S. Ostroff, and Phillip J. Brooke. Checking the Consistency of Collaboration and Class Diagrams using PVS. *RSTI - L'objet, 4^{th} Rigorous Object-Oriented Methods Workshop*, 9(4):115–134, 2003.

[28] J. Paslberg and M.J. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[29] Jean-Claude Royer. A New Set Interpretation for the Inheritance Relation and its Checking. *ACM OOPS MESSENGER*, 3(3):22–40, 1992.

[30] Jean-Claude Royer. An Operational Approach to the Semantics of Classes: Application to Type Checking. *Programming and Computer Software*, 27(3):127–147, 2002. ISSN 0361-7688, `http://www.maik.rssi.ru/index.html`.

[31] Jean-Claude Royer. Temporal Logic Verifications for UML: the Vending Machine Example. *RSTI - L'objet, 4^{th} Rigorous Object-Oriented Methods Workshop*, 9(4):73–92, 2003.

[32] A. Tsiolakis and H. Ehrig. Consistency Analysis of UML Class and Sequence Diagrams using Attributed Graph Grammars. In H. Ehrig and G. Taentzer, editors, *Proc. of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, Berlin, March 2000*, 2000. Technical Report no. 2000/2, Technical University of Berlin.

[33] N. Wirth and K. Jensen. *Pascal - User Manual and Report*. Springer, Berlin, 3 edition, 1985.

About the author

Jean-Claude Royer is currently a Professor in Ecole des Mines de Nantes, he is member of the EMN/INRIA OBASCO project and member of the LINA laboratory. His researches focus on object-oriented programming, component and formal specifications of mixed systems. He can be reached at Jean-Claude.Royer@emn.fr.