“Real World” as an Argument for Covariant Specialization in Programming and Modeling

Roland Ducournau

L.I.R.M.M., Université Montpellier 2
161, rue Ada – 34392 Montpellier cedex 5, France,
`ducournau@lirmm.fr`,
`http://www.lirmm.fr/~ducour/`

Abstract. Class specialization is undoubtedly one of the most original and powerful features of object orientation as it structures object models at all stages of software development. Unfortunately, the semantics of specialization is not defined with the same accuracy in the various fields. In programming languages, specialization is constrained by type theory and by a type safe policy, whereas its common sense semantics dates back to the Aristotelian tradition. The well known covariant vs. contravariant controversy originates here. In this paper, we investigate how modeling and programming languages deal with this mismatch. We claim that type errors are part of the real world, so they should be taken into account at all stages of software development. Modeling as well as programming languages should adopt a covariant policy.

1 Introduction

Originated in SIMULA more than 30 years ago [3], object orientation has become, by now, quite hegemonic in the field of programming languages and software engineering, not to speak of databases or knowledge representation. This hegemony has often been explained by the closeness of various object-oriented concepts to corresponding common sense notions as they have been elaborated in classic philosophy [21, 22]. Noticing that, one could hope for a *seamless* development process from so-called real world to program implementation, through analysis and design steps. However, this apparently uniform model presents some discontinuities, particularly when specialization is concerned.

Class specialization is undoubtedly one of the most original and powerful features of object orientation, yielding most of its qualities and breaking with previous programming paradigms. A large part of the literature is devoted to it, and it is the central point of many active topics of research such as inheritance (programming languages), classification or subsumption (knowledge representation), polymorphism or subtyping (type theory). Unfortunately, the semantics of specialization is not defined with the same accuracy in those various fields. Moreover, specialization may be constrained, in some field, by some external considerations. For instance, the well known *covariant vs. contravariant controversy* (e.g. [8], [18, chapter 17] or [25]) can be explained as a conflict between the demands of a type safe policy and the needs for expressivity. In this paper, we look at this well known controversy from the point of view of our common sense understanding of the "real world" and investigate whether modeling languages answer adequately to this requirement. Type errors are part of the real world. A dramatic example has been given by the "mad cow" disease: `cows`, as a specialization of `herbivorous`, should only eat `grass`, not `meat`, but it happened that they were feeded with remains of cows. So, we claim that type errors should be taken into account at all stages of software development: analysis and design methods, as well as programming languages should adopt a covariant policy.

The rest of this paper is organized as follows: section 2 briefly recalls the *de facto* standard object model, then states how specialization can be related to common sense reasoning and Aristotelian tradition and gives some hints regarding how knowledge representation formalizes it. Next section takes the viewpoint of programming languages and type theory and states the covariance vs. contravariance controversy. The case of most widely used languages is examined and some alternatives such as multiple dispatch are investigated. Section 4 looks at analysis and design methods, mainly UML, and concludes to their current abdication to impose a semantics in front of JAVA's one. In conclusion, we sketch out the specifications of a language adapted to the semantics of specialization.

2 Semantics of specialization

The *de facto* standard object model is the class-based model, consisting of *classes*, organized in a *specialization* hierarchy, and *objects* created as instances of those classes by an instantiation process. Each class is described by a set of properties, *attributes* for the state of its instances and *methods* for their behavior. Applying a method to an object follows the metaphor of *message sending* (also called *late binding*): the invoked method is selected according to the class of the object (called the *receiver*). This is the core of the model and it suffices to state the point of the specialization semantics. It is a *de facto* standard since it covers all of the widely used languages as the core of analysis and design models.

Though novel in computer science, specialization has quite ancient roots in the Aristotelian tradition, in the well known syllogism: *Socrates is a human, humans are mortals, thus Socrates is a mortal*. Here *Socrates* is an instance, *human* and *mortal* are classes. The interested reader will find in [21, 22] a deep analysis of the relationships between object orientation and Aristotle syllogistic.

2.1 Inclusion of extensions, intensions and domains

According to the Aristotelian tradition, as revised with the computer science vocabulary, one can generalize this example by saying that *instances of a class are also instances of its superclasses*. More formally, $\prec$ is the specialization relationship ($B \prec A$ means that B is a subclass of A) and Ext is a function which maps classes to the sets of their instances, their *extensions*. Then:

$$B \prec A \Longrightarrow Ext(B) \subseteq Ext(A) \qquad (1)$$

This is the essence of specialization and it has two logical consequences: inclusion of intensions (i.e. inheritance) and inclusion of properties' domains (i.e. covariant refinement). When considering the properties of a class, one must remember that they are properties of instances of the class, factorized in the class. Let B be a subclass of A: instances of B being instances of A, have all the properties of instances of A. One says that subclasses inherit properties from superclasses. More formally, Int is a function which maps classes to the sets of their properties, their *intensions*:

$$B \prec A \Longrightarrow Int(A) \subseteq Int(B) \tag{2}$$

Properties have a value in each object and can be described in the class by a *domain*, that is the set of values taken by the property in all the class's instances. For instance, the class `Person` has a property `age` whose domain is $[0, 120]$. When specializing a class, one refines the domains of inherited properties: for instance, a subclass `Child` of `Person` will have domain $[0, 12]$ for its property `age`. The function Dom maps classes and properties to sets of values. Then:

$$B \prec A \ \ \& \ \ P \in Int(A) \Longrightarrow Dom(B, p) \subseteq Dom(A, p) \tag{3}$$

The `age` example concerns attributes. Methods may have several domains, for parameters and returned value. As an example, consider classes of `Animals`, in a hierarchy à la Linnaeus, with a method `eat` defined with different domains in classes such as `herbivorous`, `carnivorous`, and so on. [18, chapitre 17] develops a longer example, more oriented towards programming languages.

The inclusions of extensions and intensions have opposite directions, while those of extensions and domains have the same: intensions can be said *contravariant* whereas domains are *covariant*, both w.r.t. extensions, i.e. specialization.

2.2 Specialization in knowledge representation

Though quite intuitive, inclusion (3) cannot be proved to be entailed by (1) without a careful definition of class extensions which needs a model-theoretic approach. Such a semantics of specialization has been formalized in knowledge representation systems called *description logics* or languages of the Kl-One family [27, 10]. In previous works, we showed that such a formalization could be exported to a more standard object model but this is not a common approach [12]. A main feature of this semantics is that the equations corresponding to (1-3) can be equivalences, not mere implications: in other words, classes can be defined as necessary and sufficient conditions and specialization between classes (then called *subsumption*) can be deduced from class properties, which leads to *classification*. Previous examples obviously need such semantics since `adult` and `child` are defined by their `age`, as well as `herbivorous` and `carnivorous` by what they `eat`. However, such a semantics is not necessarily adapted to programming languages nor to analysis and design modeling, as it has a major drawback, being essentially monotonous: one can add values, not modify them. Nevertheless, it could give some hints to precise the semantics of object models, as well as semantical bases to automatic computation of class hierarchies [13].

3 Programming languages, subtyping and polymorphism

Object-oriented programming languages can be considered as a mixture of object-oriented notions and programming languages notions. We will just consider the notion of type, central in programming languages, and focus on *statically typed* languages. Arguments in favor of static typing are numerous. The main one concerns reliability. Static, i.e. compile-time, analysis is needed to avoid dynamic, i.e. run-time, errors. Static typing allows a simple and efficient static analysis, whereas dynamic typing requires more expensive and less effective analyses. Anyway, static typing is another *de facto* standard.

3.1 Contravariance of subtyping

In a statically typed language, every entity in the program text which can be bound to a value at run-time is annotated by a type, its *static type*. At run-time, every value has a type, its *dynamic type*, i.e. the class which creates the value as its instance. In such a context, an entity is said to be *polymorphic* when it can be bound to values of distinct types, and the dynamic types of the values must *conform to* the static type of the entity. Otherwise, there is a run-time type error, which may lead to an `unknown message` error when a method, called upon this entity, is known by the static type, not by the dynamic one.

Types and classes are quite similar—a type can be seen as a set of values (extension) and a set of operators (intension)—and the conformance relationship between types, denoted by $<:$, is analogous to specialization between classes. Statically typed languages allow a static (compile-time) type error checking, i.e. a *type safe* compilation. A simple way to allow this is to define conformance through the notion of *substitutability*: a type t_2 conforms to a type t_1 iff any expression of type t_1 can be substituted by (bound to) any value of type t_2 without any run-time type error. Types can be identified with classes or, preferably, types can be associated to classes but the very point is to liken class specialization and subtyping. Class specialization can support polymorphism—an instance of a subclass can be substituted to an instance of a superclass—as long as the type of the subclass conforms to the type of the superclass. Of course, with a type safe policy. Class specialization is thus constrained by type safety.

This constraint revolves around the way types of properties can be redefined (overridden) in a subclass. Let A be a class and m a method defined in A, noted m_A. Method types are noted in a functional way, with arrow types: m_A has, for instance, type $t \rightarrow t'$. Let B a subclass of A, where m is redefined in m_B, with type $u \rightarrow u'$. The type of B conforms to the type of A, only if $u \rightarrow u'$ is a subtype of $t \rightarrow t'$. Subtyping on arrow types is defined as follows [7]:

$$u \rightarrow u' <: t \rightarrow t' \iff t <: u \;\;\&\;\; u' <: t' \qquad (4)$$

A function of type $t \rightarrow t'$ can be replaced by a function of type $u \rightarrow u'$ if the latter accepts more values as parameter ($t <: u$) and returns less values ($u' <: t'$). Following Cardelli, the return type is said *covariant*, while the parameter type is *contravariant*: this is known as the *contravariance rule*. Attribute redefinition is ruled by a mixture of them, as an attribute can be seen as a pair of two methods, a reader which returns the attribute value and a writer which set this value from its parameter's value. Thus an attribute redefinition must be both covariant and contravariant, leading to *invariance*.

3.2 Covariance vs. contravariance

Coming back to the semantics of specialization, one sees that *domain* specialization is subjected to some kind of *covariant rule* (3). The controversy lies there: *contravariance* for types versus *covariance* for domains. The fact is that domains are not types. Domains are defined as the sets of values taken by a property—i.e. an attribute, a method parameter or returned value—on all instances of a class. Static types are program annotations intended to avoid run-time type errors. Domains are ruled by existential quantifiers—there exists a value for a property—, while types are ruled by universal quantifiers—any value of the type should be substitutable. Properties may have both domains and types. Domains can be understood as subsets of type extensions [6] and there is no way to precisely express domains in programming languages, but types. Nevertheless, introducing domains in programming languages would lead to domain errors and to a *domain safe* policy, which would be essentially the same as the type safe policy. So, our thesis is that domains can be expressed by types and that domain errors are part of the real world: cows should not eat meat, but grass, and there is no way to statically check for it. One unfortunately knows that, in the real world, it has not been dynamically checked. Type safe programs are certainly more reliable, but faithful programs are better. Thus the type safe policy should be imposed only when type errors originate in the program not when they are part of the real world.

3.3 Actual languages

Actual languages are apparently ruled by the type safety dogma. C++ and JAVA, the two most widely used object-oriented languages, apply the contravariance rule, in a stricter way than needed: parameter and attribute types are actually invariant. As for return types, they are also invariant in JAVA, without any reason, but they can be covariantly redefined in C++, at least recently [15]. However, both languages present two error-proning features which make type safety unreachable. *Downcast* is a way to assume that a value of a given static type has actually a more specific subtype: this assumption must be checked at run-time, which leads to run-time type errors. *Static overloading* allows to define, in the same classes, different methods with the same name and distinct parameter types. Both features allow apparent covariant redefinition. In the case of static overloading, it remains an illusion since static overloading is ruled by a static dispatch which obviously cannot emulate dynamic dispatch but interferes with it in a very confusing way. As for downcast, it allows to precisely express the covariant semantics that a programmer would have to, at the cost of clumsiness and potential type errors. Moreover, downcast is not restricted to handling covariant parameters and can be reused as a bad general programming style.

EIFFEL is the only widely known language to rule out the contravariance rule: parameter and attribute types must be covariantly redefined [17, 18]. However, it tries to maintain the type safety dogma, with the so-called *catcall* rule, where 'cat' stands for "Changing Availability or Type". We will state the rule in the case of a type change, i.e. a covariant refinement. A *call* is *polymorphic* if the receiver's dynamic type may be different from its static type. A *catcall* is a call to a method which is covariantly redefined in the subclasses of the receiver's type. *Polymorphic catcalls are forbidden*. Unfortunately, this rule would forbid to actually use the covariantly redefined methods, if it were applicable. For the *catcall* rule cannot be implemented in separate compilation and doesn't seem to have ever been implemented. Moreover, since a global analysis is needed, the type safety could be obtained with a far less strict rule [23].

3.4 Variations around language design

Many variations around standard object model and type system have been proposed, often as an answer to this controversy. We will examine two of them: variations on method dispatch and variations on polymorphism and subtyping.

Multiple dispatch Multiple dispatch has been popularized by CLOS [4] before finding a theoretical framework in static typing [19, 8]. It has been adopted by many languages. Apart from CLOS, which is dynamically typed, none of those languages is widely used: thus multiple dispatch remains academic, despite its true interest. In standard object model, method dispatch (also called late binding) is realized according to the message sending metaphor: the type of a distinguished parameter, called the receiver, is used to select a method. Other parameters have no influence on dynamic dispatch: they are only used for static overloading, when it is the case. With multiple dispatch, all parameters are used for selecting the method. An easy way to understand multiple dispatch is to see it as a single dispatch on the cartesian product of parameter types. This is the implicit CLOS point of view. Contravariance disappears as it concerns only parameters unused for dispatch but usual modularity of classes and methods also disappears: methods are no more within classes, but between.

Another view on multiple dispatch is provided by *overloaded functions*—not to confuse with static overloading—which preserve modularity [8]. *Multimethods*, instead of methods, are associated to classes: each multimethod can have several branches, i.e. methods, which differ from each other by the types of their secondary parameters. Dispatch is then two-steps: a multimethod single dispatch on the receiver type is followed by a branch multiple dispatch on the types of other parameters (i.e. single dispatch on their cartesian product). Some typing rules allow type safe compilation. Implementing covariant refinement of parameter types is easy with multimethods. One defines a multimethod with two branches: the first one is the overriding method, with refined parameter types, while the other one has non-refined parameter types and signals a type error.

Multiple dispatch is a good programming solution, for methods only since it doesn't apply to attributes. But it is no more type safe than, either downcast in C++ and JAVA, or a truly covariant language like EIFFEL. [5] proposes a technique for automatically transform covariant programs into type safe multiple dispatch programs, by changing methods into multimethods and adding branches when meeting covariant refinement. Added branches call overridden multimethods. This technique may be adapted for pure covariant refinement: then, added branches signal a type error instead of calling overridden multimethod.

Genericity and subtyping Many types systems have been proposed, often with the aim of making EIFFEL's type system safe. Besides its covariant policy, EIFFEL's type system presents an original feature. A type can be "anchored" to the type of the receiver (`self`, `this` or `current` according to languages) with the type `like current`, also called `mytype`. The anchor can also be a property p of `self`, with the type `like` p. Anchored types are typically covariant: in the interface of a class, i.e. on a non `self` receiver, they are not type safe, unless when used as return types. Several propositions for making EIFFEL type safe have been made [9, 26]: they are mainly based on a translation of anchored types into parametric types. Indeed, as a corollary of the contravariance rule, if $A\langle T\rangle$ is a parametric type, B and C two classes, then $A\langle C\rangle$ is not a subtype of $A\langle B\rangle$ when $C <: B$, at least in a type safe policy. Thus, those propositions for making EIFFEL type safe replace an unsafe covariant specialization by a safe but non substitutable parameterization.

Another way to avoid the contravariance controversy has been to dissociate subclassing and subtyping, as in SATHER [25]. Again, the result is to allow covariant refinement, but to forbid substitutability, as with the catcall rule or EIFFEL corrections. Not surprisingly, all those approaches are more or less equivalent, since usual types A need to be defined as *recursive types* $\mu(t)A(t)$ where μ binds a variable t to the type being defined, i.e. `mytype` [1]. Though genericity (i.e. parametric polymorphism) and subtyping (i.e. inclusion polymorphism) are conceptually different, their formal bases are the same. Thus any translation of anchored types into parametric types, or any dissociation of subclassing and subtyping will have no effect on the controversy: covariant refinement, polymorphism (i.e. substitutability) and type safety are incompatible.

4 Analysis and design methods

Analysis and design methods bridge the gap between, on the one hand, real world and common sense reasoning and, on the other hand, programming languages. Clearly, analysis should be independent from programming languages. There is no evidence that analysis' specificities and goals would be very different from knowledge representation's ones. In fact, one can find bridges between them (e.g. [14]) and the main requirement for knowledge representation is that it must afford a formal support to reasoning, while analysis should only produce an informal model. So, our first point will be that analysis methods should use covariant models, as common sense reasoning and knowledge representation.

As for design methods, it is unclear how independent from programming languages they should be. Part of design methods is often dedicated to implementation, thus depends on some specific languages.

Looking at existing methods, one finds a family of universal object-oriented methods, such as OMT [24], which have been unified in UML [20]. It doesn't matter that UML is not a method but a modeling language, since we are only concerned here by the object models. At the opposite, a language like EIFFEL [17] can be seen as equipped with its own design method [18]. Not surprisingly, this method preconizes a covariant modeling, thus satisfying our demands. UML being the product of a unification process, it is presumed to have gained experience from previous methods such as OMT. Moreover, it tends towards hegemony. So it will be our main target. UML, after OMT, demands that signatures, including return types, be invariant by overriding. Moreover, in OMT, static overloading is explicitly advocated. Though one of the goals of UML is to "*support specifications that are independent of particular programming languages* ", it appears than many details of its specifications come from JAVA, such as signature invariance. A main drawback of the unifying approach of UML is then that unification applies only at lexical (i.e. entities composing a model) and syntactic (i.e. relationships between entities) levels: semantics is left to the reader or even is explicitly referred to the target programming language. In this article, we developed the case of the covariant semantics of specialization. Other features of modeling and programming languages could and should be discussed from the same point of view. For instance, the notion of visibility is realized in UML through a mixture of JAVA and C++ keywords, with a semantics which seems to be that of the programming languages, but it is well known that common keywords have different semantics in JAVA and C++ [2]. No effort seems to be made to propose a novel, proper semantics and the canonical encapsulation of SMALLTALK as well as the more complex export clauses of EIFFEL are not considered whereas they are closer to the essence of object orientation, by allowing the protection of `self`.

5 Conclusion and perspectives

A main quality of object-oriented technology is to provide to programs, through various development stages, a uniform model from common sense understanding of "real world". In computer science and technology, this is an original and priceless quality which should lead to a *seamless* development process. One could expect that object-oriented languages reflect a conflict between formal theories which do impose some technical policy, e.g. type theory for type safety, and analysis methods, which would demand some expressivity, e.g. covariant refinement. Surprisingly, there is no evidence of such a conflict. Moreover, if a controversy does exist about co and contravariance, it is confined to the programming languages community. The UML community seems to find JAVA a perfect language and doesn't seem to have ever heard about EIFFEL, not to speak of Aristotle.

Nevertheless, analysis and design methods should see themselves as programming languages' clients: they should impose their specifications of what should be a programming language, instead of adopting the specifications of the current most widely used programming languages. This is not to rule out type theory: type safety is an important viewpoint on programs, not the main one. Type errors exist in the “real world”—it could happen that cows eat meat, not grass, moreover it happened—thus, they should be integrated in analysis, design and programming stages. Programming languages designers must compromise between the languages expressivity and the formal properties they want to ensure. Such a compromise recalls the compromise between expressivity and completeness which caused a long lasting debate in the knowledge representation community [16]. But this new compromise differs for type safety is truly incompatible with the ability to express real world type errors.

Thus, our conclusion and perspectives are twofold. First, analysis and design methods, i.e. OMG, should adopt a model with covariant refinement integrating type errors both in the model and in the methods; and they should ask for programming languages adapted to such a model. Second, programming languages designers should design languages allowing an easy expression of covariant refinement, either with single or multiple dispatch; type errors should be explicitly handled at run-time, and an improved static analysis of those type errors should allow the programmer to compare the potential errors in the program with their existence in the analysis and design models. Analysts, designers and programmers should all be aware of the type errors which might happen when using covariant refinement. Moreover, if the ability to express covariance is necessary, invariance may often be sufficient: type safe models and programs must be encouraged. Languages could supply keywords to express that a specific property could or couldn’t be covariantly refined in subclasses. As a corollary, there is no solution to the controversy to expect from type theory, for two reasons. First, type errors come from the “real world”: no theory is powerful enough to modify it. Second, all the variations on types and polymorphism are mainly equivalent: a choice must always be made between specialization, polymorphism and type safety. The interested reader will find a more detailed discussion in [11]. More generally, for all features which are a matter of modeling, analysis and design methods should propose their own proper specifications to programming languages.

References

1. M. Abadi and L. Cardelli. On subtyping and matching. In W. Olthoff, editor, *Proc. ECOOP’95*, LNCS 952, pages 145–167. Springer-Verlag, 1995.
2. G. Ardourel and M. Huchard. Access graphs, another view on static access control for a better understanding and use. *J. of Object Technology*, 2002. (to appear).
3. G. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin*. Petrocelli Charter, New York (NY), USA, 1973.
4. D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common Lisp Object System specification,. *ACM SIGPLAN Notices*, 23, 1988.
5. J. Boyland and G. Castagna. Type-safe compilation of covariant specialization: a practical case. In P. Cointe, editor, *Proc. ECOOP’96*, LNCS 1098, pages 3–25. Springer-Verlag, 1996.
6. C. Capponi, J. Euzenat, and J. Gensel. Objects, types and constraints as classification schemes. In G. Ellis, R. Levinson, A. Fall, and V. Dahl, editors, *Int. Conf. on Knowledge Re-use, Storage and Efficiency (KRUSE’95)* , pages 69–73, 1995.

7. L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. McQueen, and G. Plotkin, editors, *Semantics of Data Types*, LNCS 173, pages 51–67. Springer-Verlag, Berlin, 1984.
8. G. Castagna. *Object-oriented programming: a unified foundation*. Progress in Theoretical Computer Science Series. Birkhaüser, 1997.
9. W. R. Cook. A proposal for making Eiffel type-safe. In S. Cook, editor, *Proc. ECOOP'89*, pages 57–70. Cambridge University Press, 1989.
10. F.-M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. CSLI Publications, Stanford (CA), USA, 1996.
11. R. Ducournau. "Real World" as an argument for covariant specialization in programming and modeling. RR 02-083, L.I.R.M.M., Montpellier, France, 2002.
12. R. Ducournau and G. Pavillet. Langage à objets et logique de descriptions : un schéma d'intégration. In I. Borne and R. Godin, editors, *Actes LMO'2001* in *L'Objet vol. 7*, pages 233–249. Hermès, 2001.
13. R. Godin, H. Mili, G. Mineau, R. Missaoui, A. Arfi, and T. Chau. Design of Class Hierarchies Based on Concept (Galois) Lattices. *Theory and Practice of Object Systems*, 4(2), 1998.
14. S. Greenspan, J. Mylopoulos, and A. Borgida. On formal requirements modeling languages: RML revisited. In *Int. Conf. on Software Engineering (ICSE'94)*, 1994.
15. A. Koenig. Standard – the C++ language. Report ISO/IEC 14882:1998, Information Technology Council (NCTIS), 1998. `http://www.nctis.org/cplusplus.htm`.
16. H. Levesque and R. Brachman. Expressiveness and Tractability in Knowledge Representation and Reasoning. *Computational Intelligence*, 3(2):78–93, 1987.
17. B. Meyer. *Eiffel: The Language*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK, 1992.
18. B. Meyer. *Object-Oriented Software Construction*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, second edition, 1997.
19. W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In P. America, editor, *Proc. ECOOP'91*, LNCS 512, pages 307–324. Springer-Verlag, 1991.
20. OMG. Unified Modeling Language specifications, v1.4. Technical report, Object Management Group, 2001.
21. D. Rayside and G. Campbell. An aristotelian understanding of object-oriented programming. In *Proc. OOPSLA'00*, SIGPLAN Notices, 35(10), pages 337–353. ACM Press, 2000.
22. D. Rayside and K. Kontogiannis. On the syllogistic structure of object-oriented programming. In *Proc. of ICSE'01*, pages 113–122, 2001.
23. J.-C. Royer. An Operational Approach to the Semantics of Classes: Application to Type Checking. *Programming and Computer Software*, 28(3), 2002. (to appear).
24. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1991.
25. C. Szypersky, S. Omohundro, and S. Murer. Engineering a programming language: The type and class system of Sather. In *Proc. of First Int. Conference on Programming Languages and System Architectures*, LNCS 782. Springer Verlag, 1994.
26. F. Weber. Getting class correctness and system correctness equivalent — how to get covariant right. In R. Ege, M. Singh, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems (TOOLS 8)*, pages 192–213, 1992.
27. W. Woods and J. Schmolze. The KL-ONE family. *Computers & Mathematics with Applications*, 23(2–5):133–177, 1992.