Science of Computer Programming 76 (2011) 555–586

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Metamodeling semantics of multiple inheritance

Roland Ducournau [a,*], Jean Privat [b]

[a] LIRMM – CNRS and Université Montpellier II, 161 rue Ada, 34000 Montpellier, France
[b] Dép. d'Informatique, UQAM, 210, avenue du Président-Kennedy, Montréal, QC, H2X 3Y7, Canada

ARTICLE INFO

Article history:
Received 26 December 2008
Received in revised form 5 October 2010
Accepted 21 October 2010
Available online 7 November 2010

Keywords:
Object-oriented programming
Multiple inheritance
Metamodeling
Redefinition
Linearization
Open-world assumption
Static typing
Virtual types

ABSTRACT

Inheritance provides object-oriented programming with much of its great reusability power. When inheritance is *single*, its specifications are simple and everybody roughly agrees on them. In contrast, *multiple inheritance* yields ambiguities that have prompted long-standing debates, and no two languages agree on its specifications. In this paper, we present a semantics of multiple inheritance based on metamodeling. A metamodel is proposed which distinguishes the "identity" of properties from their "values" or "implementations". It yields a clear separation between syntactic and semantic conflicts. The former can be solved in any language at the expense of a common syntactic construct, namely full name qualification. However, semantic conflicts require a programmer's decision, and the programming language must help the programmer to some extent. This paper surveys the approach based on *linearizations*, which has been studied in depth, and proposes some extensions. As it turns out that only static typing takes full advantage of the metamodel, the interaction between multiple inheritance and static typing is also considered, especially in the context of virtual types. The solutions proposed by the various languages with multiple inheritance are compared with the metamodel results. Throughout the paper, difficulties encountered under the *open-world assumption* are stressed.

1. Introduction

> Inheritance is commonly regarded as the feature that distinguishes object-oriented programming from other modern programming paradigms, but researchers rarely agree on its meaning and usage. Taivalsaari [96]

Class specialization and inheritance represent key features of object-oriented programming and modeling. Introduced in the Simula language [10], they have been related to the Aristotelian syllogistic [82–84] and contribute to the way the object-oriented approach meets software engineering requirements such as *reusability* and *extensibility*.

In spite of Taivalsaari's quotation above, inheritance is relatively simple when it is *single*, i.e. when a class cannot have more than one *direct superclass*—hence, the class specialization hierarchy is a tree or a forest. This is, however, a major limitation and there have been attempts since from the very beginning of object-oriented programming to soundly specify *multiple inheritance* in the pioneer object-oriented languages, i.e. Flavors [105], Smalltalk [12], and Simula [63]. It quickly appeared that multiple inheritance was not as simple as single inheritance since *conflicts* may occur that make the behavior hard to specify and give full meaning to the quotation. Different trends have divided the object-oriented programming community, with each one advocating a preferred policy.

* Corresponding author.
E-mail addresses: ducour@lirmm.fr, DUCOUR@LIRMM.FR (R. Ducournau), privat.jean@uqam.ca (J. Privat).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.10.006

1. A few production languages, but mainly Smalltalk [47], are in pure single inheritance, and their key feature is *dynamic typing*.
2. Several production languages, such as C++ [58,94], Eiffel [71,72], or Clos [92], propose full multiple inheritance. They were designed in the 1980s, and Python [101] is one of the few from the 1990s. All of them are widely used but have been the focus of considerable discussion, e.g. [6,91,19,104,86], and they all behave in a different way with respect to multiple inheritance.
3. Some languages are based on the close notions of *mixins* [13] or *traits* [27]. They are mostly research languages; Scala [78] is the most representative among recent ones, and Ruby [43] is one of the very few production languages that comply with this trend. A recent language, i.e. Fortress [2], is based on a close notion.
4. In the *static typing* setting, a major trend was inaugurated with Java *interfaces* [49], where classes are in single inheritance but with *multiple subtyping*, as a class can implement several unrelated interfaces; many recent languages, e.g. C# [73] and .Net languages, follow this trend.

Besides these programming languages, the main (or even only) modeling language, i.e. UML [80], includes multiple inheritance without any precise specification.

The need for multiple inheritance has also prompted a long-standing debate. Besides the aforementioned references, see for instance [88] and various related conference panel sessions. However, the fact that few statically typed languages use pure single inheritance, i.e. *single subtyping*, strongly underlines the importance of multiple inheritance. The rare counterexamples, such as Oberon [106,75], Modula-3 [50], or Ada 95 [8], result from the evolution of non-object-oriented languages. Furthermore, the absence of Java-like multiple inheritance of interfaces was viewed as a deficiency of the Ada 95 revision, and this feature was incorporated in the next version [95]. The requirement for multiple inheritance is less urgent in the dynamic typing framework; for instance, all Java multiple subtyping hierarchies can be directly defined in Smalltalk, by simply dropping all interfaces. Conversely, statically typing a Smalltalk hierarchy only involves adding new interfaces to *introduce* methods that are *introduced* by more than one Smalltalk class.[1]

Overall, despite the numerous dedicated works, multiple inheritance is not a closed issue. From this standpoint, there is no satisfactory language. As we shall see, even languages based on multiple subtyping may be flawed, since they may not ensure full reusability. In this paper, we propose a semantics of class specialization and inheritance which is "natural", or even "Aristotelian". Our proposal aims to ensure universality and simplicity by (i) clarifying concepts and clearly defining problems related to multiple inheritance in general; (ii) identifying specific multiple inheritance issues that are usually unresolved (or poorly resolved) and proposing solutions for them; and (iii) formally discussing and comparing the different specifications of many OO languages (C++, Java, Clos, Python, C#, etc.).

The proposed semantics is based on *metamodeling*, i.e. reifying the concerned entities, namely classes and properties. The proposed metamodel is the simplest metamodel that models classes and properties in such a way that each name in the program code can denote a single instance of the metamodel. It allows us to get rid of names and their associated ambiguities in order to just consider reified entities. In contrast, most object-oriented languages, especially in static typing, attempt to interpret names and inheritance in the Algol tradition, on the basis of the scope and extent—e.g. the so-called *scope resolution operator* in C++. Whereas it can work with single inheritance, i.e. the subclass is interpreted as a block nested in its superclass, it obviously fails with multiple inheritance. The first benefit drawn from this metamodel is to precisely distinguish the "identity" of a property from its "value" (or "implementation") in a given class. In turn, it strongly distinguishes between two kinds of conflict which should not be confused, even though they are currently confused in all known languages with multiple inheritance. The first kind of conflict involves property names, and occurs when a class inherits two properties with the same name or signature, which have however been introduced in unrelated classes. The second kind of conflict occurs when a class cannot choose between different implementations for the same property. A variant concerns the case where several implementations must be combined. These two categories are quite different and require different answers. The first kind is purely syntactical, and a simple unambiguous denotation would provide a solution; for instance, it would make Java fully reusable in the sense that any pair of unrelated class/interface could be specialized by a common subclass. The second kind of conflict involves the program semantics; the solution cannot rely on some syntactic feature but the languages should offer the programmer some help for managing them. One approach has been studied in depth, namely *linearization* [32,33,55,34–36,9,40,45]. These two inheritance levels were originally identified in [36], but the lack of a metamodel hindered the authors from finishing the analysis.

The approach proposed in this paper would apply to all object-oriented programming languages with multiple inheritance, multiple subtyping, or even mixins. However, it turns out that *static typing* is required to take full advantage of the metamodel. Therefore the paper is focused on statically typed languages. Without loss of generality, this proposal cross-cuts usual type theories and object calculi. Usual type theories, e.g. record types [16], are based on names, and substituting reified properties for names does not change the considered type theory. However, multiple inheritance conflicts and some related solutions have special effects on types when redefinition is not type invariant. Hence, the metamodel is also applied to *virtual types* [98], and this paper analyzes the way static typing and multiple inheritance interact.

[1] The "introduction" term is crucial here and will be more formally defined. A class *introduces* a method when it defines a method with a *new* name (or signature) that is not already defined in any of its superclasses.