Object Identity Typing: Bringing Distinction between Object Behavioural Extension and Specialization

Chitra Babu and D Janakiram
Distributed Object Systems Lab, Dept. of Computer Science & Engg.,
Indian Institute of Technology Madras,
Chennai - 600 036, India.
Email: chitra@cs.iitm.ernet.in, djram@lotus.iitm.ernet.in
URL: http://lotus.iitm.ac.in

ABSTRACT

Object Oriented Programming Languages(OOPLs) primarily allow modeling object behaviors using either class-based inheritance or prototype-based delegation. Such an approach does not make a clear distinction between the two cases of an extension to the behavior of an object versus specialization of an object behavior by another object. If an object is considered to have its own state, behavior and identity, Behavioral eXtension(BeX) of an object can be seen to retain object identity, while extending the behavior and the state. On the other hand, Behavioral Specialization(BeS) always creates a new object by specializing existing behavior. Current OOPL either model class as a type or interface as a type. Hence, these languages lack the expressiveness required to distinguish between object behavioral extension and behavioural specialization. This paper proposes modeling object identity as a type, which clearly captures this distinction. Furthermore, the proposed model ensures type-safety when various objects are composed together to achieve behavioral extension.

Keywords: Object Identity, Behavioural extension, Behavioural specialization, Denotational Semantics.

1. INTRODUCTION

Object Behavioral eXtension(BeX) and Behavioural Specialization(BeS) are two different facets of modeling object behaviors. Behavioral extension of an object is an operation that preserves the identity, whereas specialization creates an object with a new identity. Generally, in current OOPLs, inheritance is the sole mechanism available to model both BeX and BeS. While class-based OOPLs clearly capture BeS through inheritance, they lack the ability to model dynamic behavioral evolution of a single object. This is due to the fact that the behavior of an object is frozen at the time of its instantiation. Prototype-based languages capture behavioral extension through a combination of dynamic inheritance and the delegation mechanism,[1] albeit at the cost of compromising encapsulation and the safety normally provided by static typing. A clear separation in modeling object behavioral extension from behavioural specialization requires building new programming language constructs. These constructs need to address the issues of modeling object identity and type safety with respect to abstraction and encapsulation. Further, the language constructs should be built on a sound theoretical foundation keeping in mind the issues related to typing object behaviors.

Method Driven Model (MDM)[2] is an effort towards providing distinction between BeX and BeS by focusing on the issues related to object identity. The key idea behind MDM is viewing an object itself as being composed from the various encapsulated parts of a basic abstraction. MDM shares the underlying philosophy of Glue object model[3] which is the relaxation of tight coupling between abstraction and encapsulation in a systematic manner. However, it distinctly differs from Glue model in that methods are explicitly modeled as connectors. Consequently, it is possible to capture rules for breaking encapsulation and associated self rebinding.

MDM is based on the notion that an object exhibits immutable and mutable behavior. TypeMarker(TM) is an interface consisting of a set of method declarations, whose definitions are deferred. An object's mutable behaviour is captured through the TypeMarkers. The method definitions, *self* rebinding based on the communication styles, and the contracts are specified using aspects. Instead of viewing objects as rigid behavioural entities, the aspect run-time system weaves appropriate aspects along with the object to achieve BeX. Whenever an object is composed from multiple aspects, there is a possibility of the composed object's state, behavior or both being

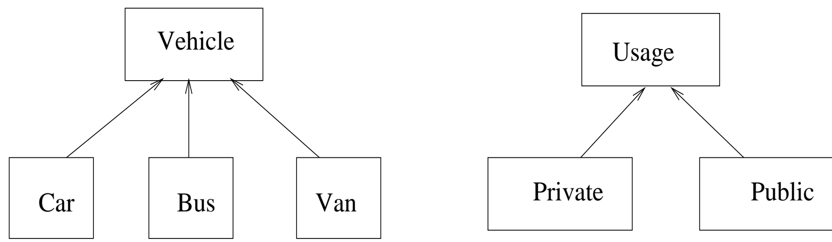

Figure 1. Transportation Problem

spread across the constituent aspects. This paper proposes a novel way of modeling the object identity as a type that captures the characteristic of identity retention by BeX.

The paper is organized as follows. Section 2 gives an example which illustrates the difficulty in modeling BeX with the current mainstream OOPLs. The various perspectives on object identity are discussed in Section 3. Section 4 analyzes some mathematical formalisms in relation to BeX and BeS. Section 5 provides an overview of MDM and explains how object identity is modeled as a type. Section 6 concludes and provides directions for future work.

2. MOTIVATION

Figure 1 illustrates a transportation domain model. In this domain, there are three specializations of the vehicle concept and two different contexts in which these specializations could be used. These contexts depict the various possible extensions to the behavior of the vehicle object. Since car, bus and taxi specialize the behaviour provided by the vehicle abstraction, that hierarchy depicts BeS. On the other hand, objects belonging to any of these categories that already have come into existence through instantiation can extend their behaviour by the introduction of public or private context. This corresponds to BeX.

One way to model this in the current class based OOPLs is to form an inheritance hierarchy as shown in Figure 2. This obviously results in unnecessary class explosion. When there are m specializations and n extensions in a given problem domain, it would be efficient to model them with $m+n$ classes instead of the mn classes that the current solution mandates. Further, the code addressing the same functionality needs to be replicated in multiple places for the various specializations. Any need to introduce additional contexts requires modification in several places. Instead, employing multiple inheritance or mixins,[4] avoids unnecessary replication of the same functionality in several classes. However, the most compelling drawback with all of these modeling approaches is that there is no way to convert a public bus or public car dynamically to a private one and vice versa.

Alternately we could model the problem by keeping two separate hierarchies as in Figure 1, and a reference to the usage class can be kept within the vehicle class. When such aggregation is used for modeling, private variables of the *vehicle* class are not accessible to the *usage* class. Further, care must be taken to bind the self reference properly to the object receiving the original message. This adds an additional burden on the programmers. Improper handling of the self rebinding can result in broken delegation problem. The complexity associated with message delivery semantics increases when messages have to be delegated across multiple levels. Such complexities can be attributed to lack of expressiveness of class based OOPLs to capture BeX properly.

3. OBJECT IDENTITY

The concept of Object Identity(OID) plays a key role in bringing distinction between the two notions BeX and BeS. This section briefly discusses the traditional view of OID. OID has been defined as that property of an object which uniquely distinguishes it from other objects.[5] OID has been studied both in the context of programming languages and databases. However, OID is different from variable names in programming languages and keys in databases. Implementation concepts such as surrogates represent system generated globally unique identifiers

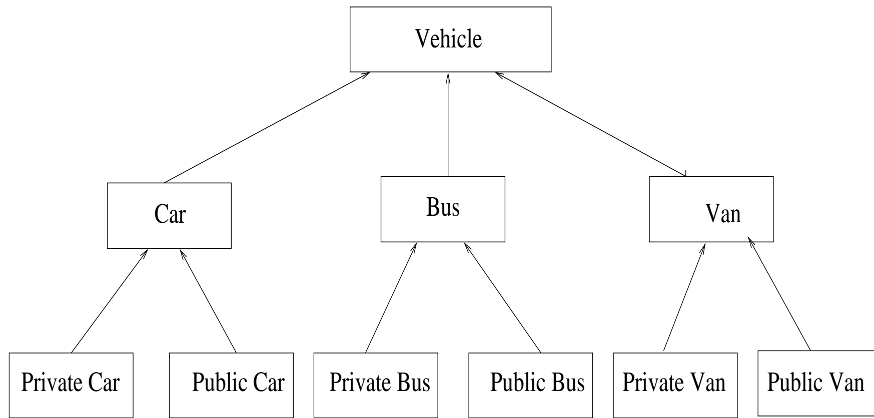

Figure 2. Inheritance Hierarchy

for objects. In all these contexts, the main focus is on physically locating the objects without any ambiguity. From this perspective, Wegner[6] argues that OID should not be tied down with the object's attributes, name, behavior or the address at which the object resides. This is because, two objects can have identical attributes, or identical behavior. Further, an object may have aliases and an object can even migrate. Bearing all these in mind, Wieringa[7] came up with "Singular reference" and "Singular naming" requirements for the OID naming scheme. These requirements mandate that at every possible state, each name refers to exactly one object and each object is referred to by exactly one name. In addition to this, "Rigid referencing" and "Rigid naming" requirements[8] were also imposed to avoid any reuse of OIDs and to exclude renaming of objects. Mendelzon[9] contends that OID should remain the same irrespective of any change in the object's behavior, (i.e) even when an object changes its class.

Nevertheless, the inherent essence of identity cannot be captured, when unique identifiers are assigned at the time of object creation. The identity of an object can change dynamically during program execution. In order to capture the key perspectives of object identity, in the present work, various mathematical formalisms have been examined in detail. The next section briefly discusses the identity and typing related issues as seen by these formalisms.

4. MATHEMATICAL FORMALISMS

4.1. λ Calculus

λ Calculus is a mathematical system that has been widely used in the specification of programming language features, and in the study of type systems. Typed λ calculus augmented with quantification operators helps in capturing certain OO language notions. From the work of Cardelli et al.,[10, 11] it is evident that parametric and inclusion polymorphisms can be modeled by universal and bounded universal quantification respectively. Further, it also shows that data abstraction and information hiding can be represented by existential quantification. This work does not distinguish between conceptual sub-typing and the inheritance mechanism. However, under practical situations, inheritance is used both as a conceptual specialization mechanism and as a vehicle for code reuse. The latter use of inheritance cannot be explained by λ calculus. Further, the notion of *self* also cannot be captured in this formalism. In order to address this need, Cardelli et al.[11] proposed ζ calculus for objects. This approach integrates the *self* semantics based on the fixed point theory into calculus. In both λ and ζ calculi, BeS alone is captured and there are no easy means to explain BeX.

4.2. Algebra

The term *algebra* denotes abstract behavior of a class of objects. An algebra consists of sets of data together with some functions that operate on them. The traditional algebra has been generalized to many sorted algebra[12] to

model abstract data types whose interface may include procedures which take arguments from more than one domain. The interface and corresponding implementation are captured by signature and its associated algebra.

Francesco et al.[13] proposed a formal model of class using algebraic specification. Using this model, a clear distinction has been made between the conceptual inheritance based on "is-a" link, and the implementation inheritance.

The hidden sorted Order Sorted Algebra(OSA)[14] extends the classical treatment of abstract data types to the notion of state. The possible internal states of an object are the elements of a "hidden" sort. Encapsulation can be captured using these hidden sorts.

The notion of an OSA, introduced initially by Goguen et al.[15] models subtypes and inheritance. An order-sorted signature is a many-sorted signature with an ordering relation $\leq$ on its sorts. Thus, the traditional concept of individual algebras has been extended to systems of related algebras. This enhancement permits BeS to be modeled, but not BeX.

4.3. Denotational Semantics

Denotational Semantics(DS)[16] is a technique for describing the meaning of programs in terms of mathematical functions on programs and program components. Cook and Palsberg[17] proposed that, objects are modeled as record values with their fields representing methods. Records can be viewed as functions from a domain of labels to a heterogeneous domain of values. A *generator* function defines a class. The Least Fixed Point(LFP) of the "generator" formally explains an object, since an object itself is self-referential. The modification component that differentiates the derived class from the base class is expressed as a *wrapper* function of two arguments, one representing *self* and the other representing *super*. *Wrapper application* mechanism is used to change the self-reference in inherited methods. A *wrapper* is applied to a *generator* to produce a new *generator* by first distributing *self* to both the *wrapper* and the original *generator*. Thus, DS captures BeS.

An explanation of DS for prototype-based dynamic inheritance has been provided by Steyeart et al..[18] In this case, objects need a changing version of themselves rather than the fixed versions. Hence, objects are modeled as generators instead of as fixed points to generators. Further, message passing requires that the message receiver be properly wrapped every time so that self reference would be set appropriately. This can be regarded as an explanation for BeX in the context of prototype based languages. However, it does not explain BeX as a notion that retains the underlying object's identity in the context of class-based OOPLs.

5. METHODOLOGY AND FORMALISM

5.1. Overview of MDM

The essential concept underlying MDM is viewing objects as composition of various encapsulated parts of a basic abstraction. The philosophy behind modeling methods as connectors between objects is to facilitate specification of rules for systematic infringement of encapsulation, and related self rebinding semantics. These factors are specified using aspects. In this context, the "aspect" should not be viewed in the traditional way of modeling a cross-cutting concern. In MDM, different factors related to a given class are separately captured using aspects. The state of the object at any point in its life-cycle dictates the set of aspects that should be weaved with the class. The fundamental entity in this model is a partial class, which comprises of:

- **Composable-Aspects:** The set of aspects that can be weaved with the partial class,
- **Aspect-View:** Visibility of the private attributes to other aspects,
- Private, Protected and Public attributes,
- Methods describing the fixed behavior,
- **TypeMarker(TM):** Declaration of methods describing the variable behavior.

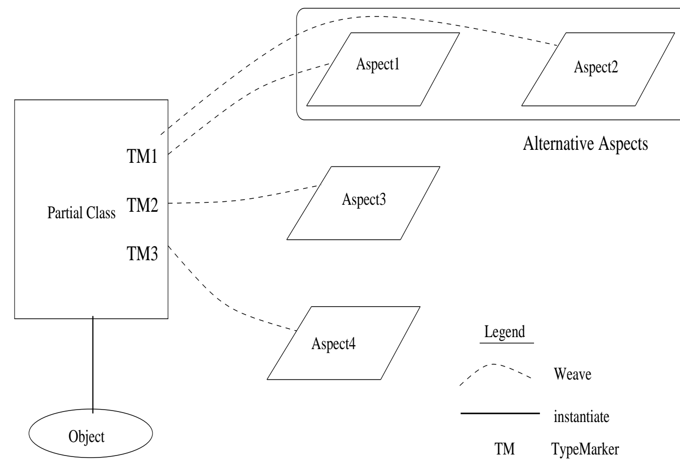

Figure 3. Class Design in MDM

The class is called as "partial" because it becomes complete only after the proper TM method definitions are weaved based on the state of the object.

Various considerations such as encapsulation, communication styles, contractual obligations, identity semantics and others are captured using the construct **aspect**. This consists of the following units:

- **Unit-Type**: This unit specifies the category of the TypeMarker which can be one of "Part-of", "using", "in" or "out", and communication styles such as "delegation" or "consultation".
- **Unit-Encap:** This unit defines the TM methods of the partial class. Inside this unit, **replace** keyword signifies the definition of TM methods of category "Part-of" or "Using". The keywords **before** and **after** are used for specifying additional actions associated with TM methods that fall under "in" category.
- **Unit-Identity:** The rebinding of self can be specified by the programmer according to the specific problem needs.
- **Unit-Contract:** This unit specifies:
  - the pre-conditions, which must be satisfied for this aspect to be weaved with an object
  - the post-conditions that must hold before the aspect gets unweaved
  - interdependencies among the TMs specified as an invariant
  - the preconditions that need to be checked before a particular method execution
  - the post-conditions that a given method must ensure at the end of its execution
- **Unit-Style:** This unit specifies the styles in which the objects should be composed, such as **Pipe and filter**, **event** etc.

Figure 3 illustrates the construction of a class in its entirety through weaving of suitable aspects along with the partial class at the control points specified by the TypeMarkers.

Aspects are also instantiable analogous to classes. Based on the state of the object during its life-cycle, the aspect run-time system dynamically weaves the appropriate sets of aspects along with it. Further details regarding the model can be found in Babu et al..[2]

5.2. Typing Object Identity

Types, in general, help to enforce correctness of programs by imposing appropriate constraints. The fundamental objective of a well-defined type-system is to minimize the possibility of run-time errors during program execution. Initially, procedural languages modeled data alone as type. Later, in the context of object-orientation, some programming languages treat the concepts of class and type as being identical. Class serves as a template for defining both structure and behavior. Any object instantiated from a given class conforms to the structure and behavior dictated by that class. Languages such as Java[19] model behavior alone as type. This section discusses a novel approach of modeling the object identity as a type so that the distinction between the two notions of BeX and BeS can be formally explained.

Traditionally, whenever an object is created, it is assigned a unique identity by which it can be referred. An instance created from a single class or a statically defined class composition structure will map to a single unique identity. On the other hand, if an object is dynamically composed out of multiple aspects, as in MDM, the composed object's identity is an "AND" of identities of the object and all the constituent aspects. However, the phenomenon of behavioral extension of an object needs to retain the underlying object's identity all the time. This has been achieved in the present work by modeling OID as a type. This approach involves two levels of abstraction: one level for defining the type and another for defining the actual identity of the object.

At the first level, OID is captured as a type (i.e) a set containing the name of the partial class and composition of names for each possible partial class-aspect combination. This set is known as OID_Type. At the second level, objects instantiated from this OID_Type are assigned object identities each of which comprises a set of elements. This set is called $OID_Instance$. The elements of this set are the identity of the partial object and tuples corresponding to identities of partial object and appropriate aspect instances. In this context, the identity of the partial object is the "self" which is defined through denotational semantics by finding the fixed point of the generator function corresponding to the partial class.

Applying this approach to the example discussed in section 2, car, bus and taxi are modeled as partial classes with a TM for public/private functionality. Two different aspects capture the behaviour corresponding to public and private vehicles respectively. MDM makes it viable to model the same physical vehicle as public or private through weaving appropriate behaviour based on the conditions captured in the "Unit-Contract" section of aspects. Identity-Type for car is as shown below:

$$OID_Type\ \ Car = \{PC_Car, < PC_Car, PublicAspect >, < PC_Car, PrivateAspect >\}$$

where PC_Car is the name of the partial class associated with car, and PublicAspect and PrivateAspect are the names of the corresponding aspects. The identity of a particular car object is defined as:

$$\begin{aligned} OID_Instance\ \ myCar &= \{Self_myCar, < Self_myCar, Self_PublicAspin >, \\ &\quad < Self_myCar, Self_PrivateAspin >\} \end{aligned}$$

in which $Self_myCar$, $Self_PublicAspin$, and $Self_PrivateAspin$ are *self*s of the *myCar* object, *PublicAspect* and *PrivateAspect* instances respectively. Whenever *myCar* object acquires public or private behaviour, its identity is governed by the appropriate identity tuple in the set. Nevertheless, these tuples belong to the same set and hence the preservation of identity is explained at the set level. Since the set OID_Type is built from TM information, and only those aspects which conform to TM interface are taken into account in constructing the tuples, type safety is guaranteed.

Modeling OID as a type uniformly captures both BeX and BeS. Whenever BeS occurs, the sets OID_Type and $OID_Instance$ themselves will change for the specialized object. On the other hand, when the behavior of an object is extended, the sets OID_Type and $OID_Instance$ will remain the same for the extended object. However, the value of the identity type will switch among the elements of the set, based on the current class-aspect structure that dictates the extended object's state and behavior. Further, the object identity is also governed by the specific tuple of *self* of object instantiated from the partial class and *self*'s of the aspect instances.

6. RELATED WORK

Predicate classes[20] proposed by Chambers et al. are similar to regular classes except that they have an additional predicate expression associated with them. This allows the appropriate dispatch of the multi-method[21] when the state of the object is changed which is captured by the predicate expression. Even though predicate classes support dynamic reclassification of objects, based on the run-time state of the objects, it cannot respond to changes in external environments in which the objects function. The reclassification is also lazily done, whenever the particular multi- method is invoked. On the other hand, in MDM, the state change is implicitly monitored and the appropriate aspect is weaved with the object instantiated from partial class, thus changing the class structure that dictates the object's current behaviour. Since predicate classes dynamically change the inheritance hierarchy, it leads to ambiguity. In contrast, MDM strives for orthogonality through usage of inheritance only for BeS. Further, MDM allows the specification of postconditions too in the contract-section, which enables catching possible exception conditions and taking appropriate actions.

Fickle[22] is a language developed by Drossopoulou et al., with the objective of dynamically re-classifying an object, while preserving its identity. The language uses two types of classes known as state classes that represent object's possible states and root classes that define the commonalities among these state classes. State classes targeted for reclassification are depicted as subclasses of root classes. Thus, *Fickle* also uses inheritance to capture BeX, which is precisely what MDM intends to avoid. Further, the re-classification should be done in *Fickle* through explicit language constructs inside the method body, while the change in class structure is transparent in MDM.

Balloon types[23] enforce strong encapsulation by ensuring that no state reachable either directly or transitively by a balloon object is referenced by any external object. Even though partial class and its associated aspects together enforce strong encapsulation at the level of the composed object, the objective of MDM is completely different from that of Balloon types. While balloon types address the problem that arise from aliasing through specifying the ability to share state as a first class type, MDM aims to distinguish BeX and BeS clearly through typing object identity.

Rondo object model[24] introduces an additional abstraction known as *classcombiner* in between classes and objects. This model also uses denotational semantics for explaining the essence of the model in a formal way. In Rondo, an object is a fixed point of a generator function corresponding to the classcombiner, which is obtained by combining the generator functions representative of the individual classes that are part of the classcombiner. Hence, the identity of the object is not preserved while it extends its behaviour which blurs the differences between BeS and BeX. In contrast, since MDM models an object as a set of tuples, where each tuple corresponds to the *self*s of object instantiated from the partial class and the current active aspects that are applicable, the identity preservation after BeX is properly captured.

7. CONCLUSIONS

The need to distinguish between object behavioral extension and specialization, from the viewpoint of object identity has been identified. Various mathematical formalisms currently in use have been examined and shown to be inadequate to capture the notion of BeX. Modeling object identity as a type, as proposed in this paper allows both BeX and BeS to be captured equally well. The key idea is to view an object as an instance of the OID_Type, rather than as an instance of a single class. This approach clearly brings out the fact that BeX is an identity preserving operation while BeS is an identity altering one. Object identity typing has enormous potential in distributed object frameworks for generating OIDs that reflect the inherent essence of objects as opposed to identifiers which bear no semantic connection to objects.

REFERENCES

1. H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems," in *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA)*, pp. 214–223, Sept. 1986.

2. C. Babu and D. Janakiram, "Method Driven Model: A Unified Model for an Object Composition Language," Tech. Rep. IITM-CSE-DOS-04-05, Indian Institute of Technology, Madras, India, 2004. Aceepted for publication in ACM SIGPLAN Notices.
3. D. J. Ram and O. Ramakrishna, "The Glue Model for Reuse by Customization in Object-Oriented Systems," Tech. Rep. IITM-CSE-DOS-98-02, Indian Institute of Technology, Madras, India, 1998.
4. G. Bracha and W. Cook, "Mixin-Based Inheritance," in *Proceedings of the Fourth ACM Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA)*, Oct. 1989.
5. S. N. Khoshafian and G. P. Copeland, "Object Identity," in *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA)*, pp. 406–416, Sept. 1986.
6. P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," *ACM OOPS Messenger* , pp. 7–87, June 1990.
7. R. Wieringa and W. D. Jonge, "The Identification of Objects and Roles - Object Identifiers Revisited," Tech. Rep. IR-267, Vrije Universiteit, Amsterdam, 1992.
8. R. Wieringa and W. De Jonge, "Object Identifiers, Keys, Surrogates - Object Identifiers Revisited," *Theory and Practice of Object Systems* **1**(2), pp. 101–114, 1995.
9. A. O. Mendelzon and T. Milo and E. Waller, "Object Migration," in *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems(PODS) Conference*, pp. 232–242, 1994.
10. L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction and Polymorphism," *ACM Computing Surveys* **17**, pp. 471–522, Dec. 1985.
11. M. Abadi and L. Cardelli, *A Theory of Objects*, Springer-Verlag, 1996.
12. P. Wegner, "The Object-Oriented Classification Paradigm," in *Research Directions in Object-Oriented programming*, B. Shriver and P. Wegner, eds., The MIT Press, Cambridge, Massachusetts, 1987.
13. F. Parisi-Presicce and A. Pierantonio, "An Algebraic Theory of Class Specification," *ACM Transactions on Software Engineering and Methodology* **3**, pp. 166–199, Apr. 1994.
14. J. Goguen and D. Malcolm, "A Hidden Agenda," *Theoretical Computer Science* **245**(1), pp. 55–101, 2000.
15. J. Goguen and R. Diaconescu, "An Oxford Survey of Order Sorted Algebra," *Mathematical structures in computer science* , 1994.
16. R. Tennent, "The Denotational Semantics of Programming Languages," *Communications of the ACM* **19**, pp. 437–453, Aug. 1976.
17. W. Cook and J. Palsberg, "A Denotational Semantics of Inheritance and its Correctness," in *Proceedings of the Fourth ACM Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA)*, pp. 433–443, Oct. 1989.
18. P. Steyaert and W. De Meuter, "A Marriage of Class- and Object-Based Inheritance Without Unwanted Children," in *Proceedings of the ECOOP 1995 Conference*, pp. 127–145, 1995.
19. K. Arnold and J. Gosling, *The Java programming Language*, Addison-Wesley, 2000.
20. C. Chambers, " Predicate Classes," in *Proceedings of ECOOP 93*, pp. 268–297, 1993.
21. C. Chambers, "Object-Oriented Multi-Methods in Cecil," in *Proceedings of the European Conference on Object-Oriented Programming*, July 1992.
22. S. DrossoPoulou and F. Damiani and M. Dezani-Ciancaglini and P. Giannini, "More Dynamic Object Re-Classification: Fickle II," *ACM Transactions on rogramming Languages and Systems* **24**(2), pp. 153–191, 2002.
23. P. G. Almeida, "Balloon Types: Controlling Sharing of State in Data Types," in *Proceedings of ECOOP 97*, pp. 32–59, 1997.
24. M. Mezini, "Dynamic Object Evolution Without Name Collision," in *Proceedings of the European Conference on Object Oriented Programming*, pp. 191–217, 1997.