MASPEGHI 2004 Mechanisms for Speialization, Generalization and Inheritance

Philippe Lahire, Gabriela Arévalo, Hernán Astudillo, Andrew P. Black, Erik Ernst, Marianne Huchard, Markku Sakkinen, Petko Valtchev

HAL Id: lirmm-00109148

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00109148

Submitted on 24 Oct 2006

Proceedings of The 3rd International Workshop on MechAnisms for SPEcialization, Generalization and inHerItance

MASPEGHI'04

Oslo, Norway, 15 June 2004

At ECOOP 2004, 14 - 18 June 2004

Ph. Lahire, G. Arévalo, H. Astudillo, A.P. Black, E. Ernst,

M. Huchard, M. Sakkinen, P. Valtchev (eds.)

Organization and Program Committee

Philippe Lahire (*primary contact*)
Laboratoire d'Informatique Signaux et Systèmes de Sophia Antipolis (I3S),
Bâtiment Euclide, Les Algorithmes, 2000 route des Lucioles, BP 121,
06903 Sophia Antipolis Cedex, France.
E-mail: *Philippe.Lahire@unice.fr*
Url : *http://www.i3s.unice.fr/~lahire*

Gabriela Arévalo
Software Composition Group
Institut für Informatikund angewandte Mathematik, Neubrückstrasse 10,
3012 Bern, Switzerland
E-mail: arevalo@iam.unibe.ch
Url: http://www.iam.unibe.ch/~arevalo/

Hernán Astudillo
Departamento de Informática, Universidad Técnica Federico Santa María
Valparaíso, Chile.
E-mail: *hernan@acm.org*
Url : *http://www.ime.usp.br/~ha/*

Andrew P. Black
Department of Computer Science & Engineering,
OGI School of Science & Engineering, Oregon Health & Science University
20000 NW Walker Road, Beaverton,
OR 97229, USA.
Email: *black@cse.ogi.edu*
Url: *http://www.cse.ogi.edu/~black/*

Erik Ernst
Department of Computer Science, University of Aarhus,
Åbogade 34, DK-8200 Århus N, Denmark.
E-mail: *eernst@daimi.au.dk*
Url : *http://www.daimi.au.dk/~eernst/*

Marianne Huchard
Laboratoire d'Informatique, de Robotique et Microelectronique de Montpellier (LIRMM),
161, rue Ada, 34392 Montpellier cedex 5, France.
E-mail: *huchard@lirmm.fr*
Url : *http://www.lirmm.fr/~huchard/*

Markku Sakkinen
Department of Computer Science and Information Systems,
P.O.Box 35 (Agora), FIN-40014, University of Jyväskylä, Finland.
E-mail: *sakkinen@cs.jyu.fi*
Url : *http://www.cs.jyu.fi/~sakkinen/*

Petko Valtchev
Département d'Informatique et recherche opérationnelle (DIRO),
Université de Montréal, CP 6128, Succ. Centre-Ville,
Montréal, Québec, Canada, H3C 3J7.
E-mail: *petko.valtchev@umontreal.ca*
Url : *http://www.iro.umontreal.ca/~valtchev/*

About the Workshop

This workshop may be seen as an heir to the past versions of MASPEGHI (at *OOIS 2002* and ASE *2003*) and of the Inheritance Workshop (at *ECOOP 2002*). This is multiple inheritance between workshops, and so we intend to apply the results of the previous workshops to the organization of this one, while at the same time making it possible to reuse the results of this workshop in the next one (MASPEGHI 2005). MASPEGHI 2004 will continue the discussion about mechanisms for managing specialization and generalization of programming language components: inheritance and reverse inheritance, specialization and generalization, and other forms of inheritance, multiple, single, mixin or trait-based. The scope of the workshop reflects two main concerns: *(i)* the various uses of inheritance, and *(ii)* the difficulties of implementation and control of inheritance in practical applications. Different communities, such as the design methods, database, knowledge representation, data mining, object programming language and modelling communities, address these concerns in different ways. Thus, one of our goals is to bring together a diverse set of participants to compare and contrast the use, implementation and control of inheritance as practiced in these communities.
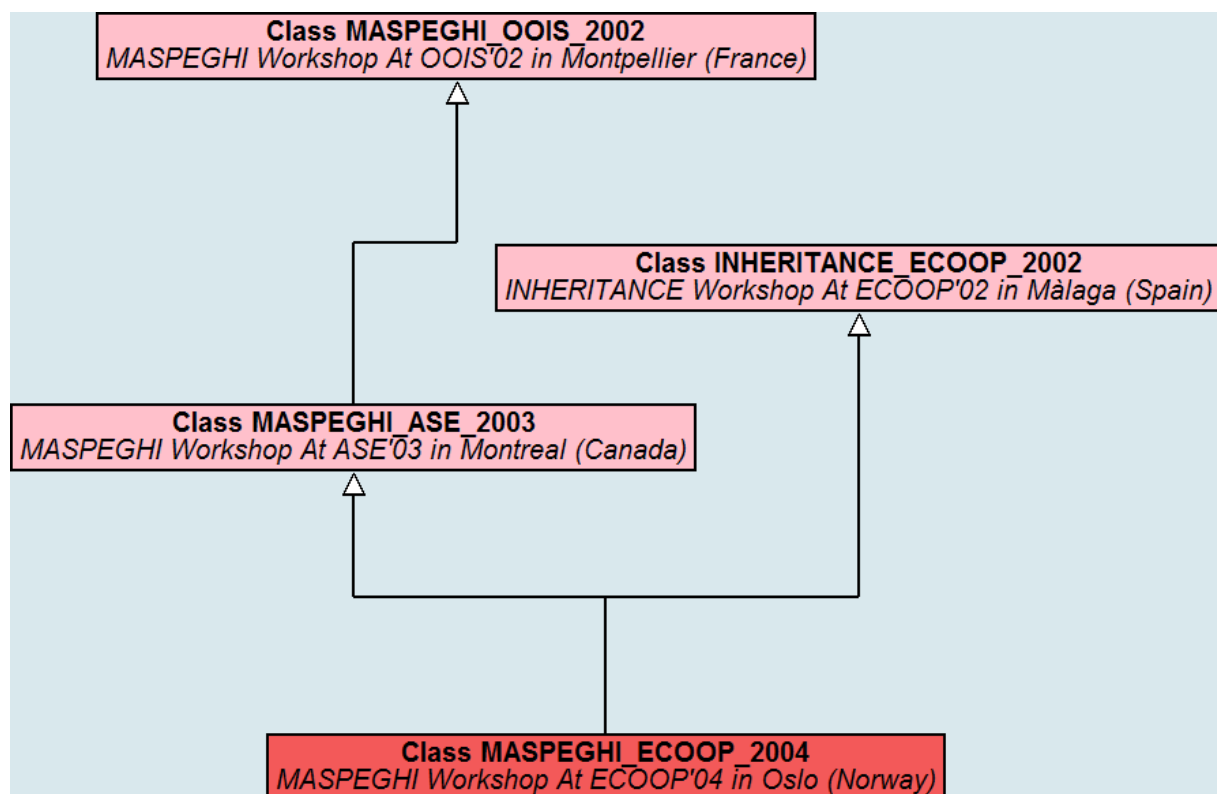
MASPEGHI 2004 will continue the discussion about mechanisms for managing and manipulating specialization and generalization hierarchies: inheritance and reverse inheritance, specialization and generalization, interface and implementation inheritance, multiple, single, mixin and trait-based inheritance, etc. We are concerned with both the uses of inheritance, and the difficulties of implementing and controlling it.

These concerns are reflected differently by disciplines such as databases, knowledge discovery and representation, modelling and design methods, object programming languages, with emphasis put either on problem domain modelling or on organizing the computational artefacts that simulate the

domain. For example, in knowledge representation, the modelling role of classes prevails: hierarchies are repositories of validated knowledge, which support the acquisition of new knowledge. In analysis and design, the purpose of the hierarchy shifts as the design matures from modelling to organizing. Hence, modern OOA&D methods support the gradual evolution of class hierarchies from one use to the other.

Despite the wide use of specialization hierarchies, there is no standard methodology for constructing and maintaining them independently from the domains that they represent and the artefacts that they organize. We hope that this workshop will provide participants a way to learn from each other and work together to develop such a methodology.

The Worksop organisers
Gabriela Arévalo,
Hernán Astudillo,
Andrew P. Black,
Erik Ernst,
Marianne Huchard,
Philippe Lahire,
Markku Sakkinen,
Petko Valtchev

Workshop Organization

This workshop is organized in four sessions:
- Session 1: Form and Transform, Dealing with evolution
- Session 2: Class composition
- Session 3: Different kinds of subclassing relationships
- Session 4: Contradiction between desired subtyping/specialization relation and language mechanism

Here are some more details about the topics addressed by these sessions.

Form and Transform, Dealing with evolution

In this topic, we would like to explore the way methodologies, languages and tools can help us or at the contrary bother when dealing with hierarchy construction and evolution. The papers of the session more precisely raise two questions: Is it realistic to imagine that automated procedures will guide or even replace expertise of human designers? Which type of help can we expect from these automatic tools? Would the use of automatic tools lead to uniformly-shaped (normalized) hierarchies? Is such a kind of normalization desirable or not?

Some forms of evolution (like inserting classes in the middle of the hierarchy, or new features to classes already in use) are really difficult with standard languages without changing other parts in user programs. Nevertheless they are necessary in many situations because requirements and knowledge of the domain evolve. Is it a good thing to admit such forms of evolution? Discuss language features which would admit such forms of evolution.

Class composition

Is class composition worthwhile? Pro: it is powerful. Con: the resulting software is complex and hard to maintain.

Different kinds of subclassing relationships

In UML it is possible to provide a more accurate definition of the kind of inheritance to use through the specification of tagged values and/or UML profiles. Should we use this facility when we design an application? How many kinds of inheritance relationships are needed? If we use several kinds of relationships is it suitable to get also several kinds of inheritance relationships within programming languages or systems? How many kinds does your language/system/.. have or should have according to your point of view?

Contradiction between desired subtyping/specialization relation and language mechanism

What to do when the desired subtyping relation that one would like to exploit in one's program does not match the particular subclassing mechanism that one has employed to create that program? For example programming, inheritance is often employed as a reuse mechanism, with no intent to create a subtype. And subtypes can be built through many mechanisms other than inheritance. What facilities should languages offer to deal with this distinction, and how can existing languages be used to address it?

Table of Contents

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
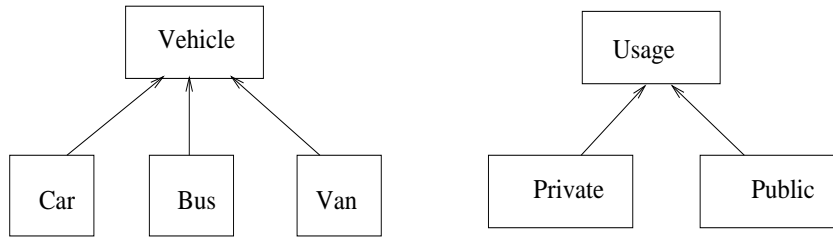
1

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
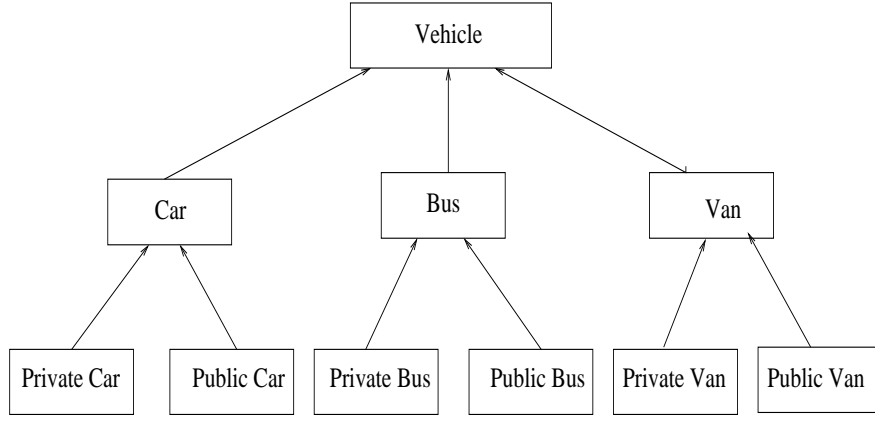
Vehicle

Car　　Bus　　Van

Private Car　Public Car　Private Bus　Public Bus　Private Van　Public Van

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

3

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.
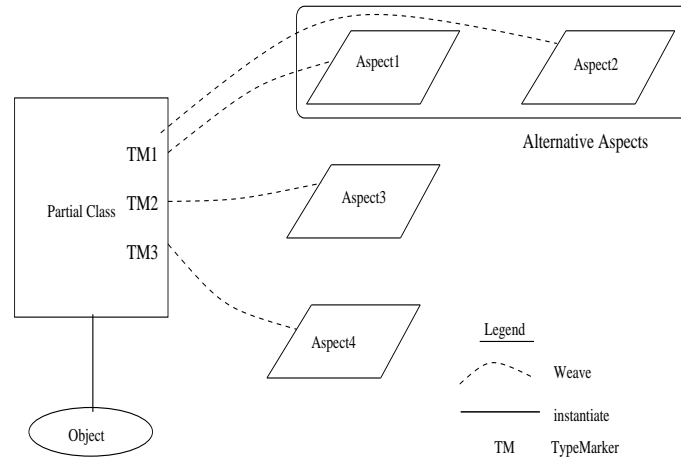
Figure 3. Class Design in MDM

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

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:

$$OID\_Instance\ myCar\ =\ \{Self\_myCar, < Self\_myCar, Self\_PublicAspin >,$$
$$< Self\_myCar, Self\_PrivateAspin >\}$$

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

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.

A Reverse Inheritance Relationship for Improving Reusability and Evolution:
The Point of View of Feature Factorization

Ciprian-Bogdan Chirila[*], Pierre Crescenzo[**], and Philippe Lahire[**]

[*] University Politehnica of Timişoara, Romania,
chirila@cs.utt.ro
[**] University of Nice-Sophia Antipolis, France
Pierre.Crescenzo@unice.fr, Philippe.Lahire@unice.fr

ABSTRACT

Inheritance is one important and controversial issue of object-oriented programming, because of it's different implementations and domain uses: design methods, database, knowledge representation, data mining, object programming languages, modelling . . .

Most of the object-oriented programming languages have a direct implementation of specialization, thus we promote the idea that a relationship between classes based on generalization can help in the process of reuse, adaptation, limited evolution of class hierarchies. We name it *reverse inheritance.*

Our goal is to show that reverse inheritance class relationship and it's supporting mechanisms can be used to accomplish the objectives mentioned earlier. Another goal is to prove the feasibility of the approach. On the other hand we analyze some use cases on how the objectives are reached.

Keywords: reverse inheritance, factoring, reuse, adaptation, limited evolution

1. INTRODUCTION

Inheritance is the mechanism used in object-oriented languages to specialize and to adapt the behaviour of a class. It is the backbone of any object-oriented system. As many implementations for inheritance exist, as many object-oriented programming paradigms[3] may be considered.

Inheritance notably offers a way to share (to factor) common features (attributes and methods) between classes, leading to hierarchies without multiple declarations of the same feature.[2]

In our approach we propose to use *reverse inheritance* relationship between classes to improve software reusability (to adapt it according to the context) and to address small evolution or refactoring. Reverse inheritance can be particularly useful when we want to reuse class hierarchies that are developed independently in different contexts. Following this idea, we would like to use reverse inheritance in order to implement a limited way to perform separation of concerns. It is important to note that we address in this paper reverse inheritance in the framework of a language which supports only single inheritance like Java. Impact of multiple inheritance and assertions will be studied but they are out of the scope of this paper.

One of our previous reports[1] proposes a set of features which should be associated to reverse inheritance in order to address the objectives mentioned above. These features are dealing with the insertion of new methods or attributes, their factorization, renaming or redefinition and the access to the code of the descendant. Each of these features must be studied into details and this paper is a first attempt for the description of the main issues related to factorization and renaming. This report addresses also the main uses of reverse inheritance such as inserting a class into a hierarchy, to link two hierarchies, etc. Moreover it describes another use of reverse inheritance which is mainly related to the specification of class hierarchy refactoring. This implied that the use of reverse inheritance is volatile and that the only relationships that persists are inheritance relationships. Objectives of this paper show that it is not the type of uses that we address from now.

The paper is organized in the following way: The second section presents some of the main aspects of the factoring mechanisms according to the state of the art. In the third section we propose a possible syntax and

implementation directions are provided for the factoring mechanism. The fourth section addresses more especially the handling of signature matching and adaptation (syntax and feasibility). Section five draws the conclusions of our study and states the future works.

2. TOWARDS THE DEFINITION OF THE FACTORING MECHANISM

As it has been said in the introduction, we focus on the *factoring mechanism* which works along with other supporting mechanisms like: feature adding, descendant access or renaming.[1] The factoring mechanism in the context of reverse inheritance class relationship relies on the relocation of methods and attributes from classes to their superclasses.

There are many reasons for factoring a class hierarchy: i) multiple occurrences of a method along the inheritance tree means an overhead to the calling mechanism because of the name resolution conflict[2]; ii) multiple declared fields along the inheritance path induce multiple redundant modifications of its occurrences[2]; iii) it is more natural to define concrete subclasses and then to extract commonalities into superclasses.[6]

Sakkinen [7] discusses Pedersen's approach of factorization, which involves the factorization of features from one selected, principal subclass. He proposes that the programmer should specify for each method from which subclass it should be factored.

Moreover [4] defines the features which are common to a set of classes: these features must have the same name in each class or are subject to be renamed. With this approach it is possible to define a signature to which corresponding signatures in each class must conform. This signature may also contain precondition (respectively postcondition) that must not be weaker (respectively stronger) than the ones associated to the methods to be factored. For the common features also, it should be possible to define a precondition other than False which is not weaker than the precondition for the feature in each class [*].

In [2] the authors analyze algorithms, based on Galois sub-hierarchies. They are applied to hierarchies in order to find an ideal factorization from the point of view of minimizing the number of feature declarations. They use metrics to count the number of occurrences of redundant features and propose algorithms for restructuring in order to build an optimal hierarchy.

In the approach of [5] which deal with refactoring, they propose a factorization methodology which modifies the code but preserves it's original behaviour. The methodology consists in isolating common features and code, and creating abstract superclasses, based on the following steps: i) to add function signatures to superclasses, ii) to make function bodies compatible, iii) to move variables and to migrate common behaviour to the abstract superclass.

In our approach the reverse inheritance relationship is used as a mean to increase reusability, so that it is close from Pedersen and Sakkinen approaches that integrates it as a basic language mechanism. We consider that features have to be factored in the following manner: common attributes have to be moved from subclasses into superclasses and common signatures of methods have to be copied into superclasses, creating new abstract methods [†]. Our approach compared with [2] is not automatic. We explore also the possibility to adapt (when it is meaningful) the signature of non-matching features when they have the same semantics (see section 4). In figure 1 the two classes contain a common attribute (*attribute1*) and a common method (*method1()*) which are factored. For the reverse inheritance class relationship we proposed the use of a new keyword *infers* in the definition of the new superclass. Situations like the one presented in our example, with factored features having the same signatures, are quite rare and context dependant. Real situations dealing with method having slightly different signatures but that should be factored must be analyzed and solved. In our approach, the problems related to factorization which are discussed in the next sections are the following: i) how to identify the feature that should be factored - signature matching and, ii) how to adapt these features in order to match the signature specified within the superclass - signature adaptation.

---

[*]This approach is related to Eiffel.

[†]The impact of access modifiers like *public*, *protected* or *private* is not discussed in this paper but it is taken into account by the approach.
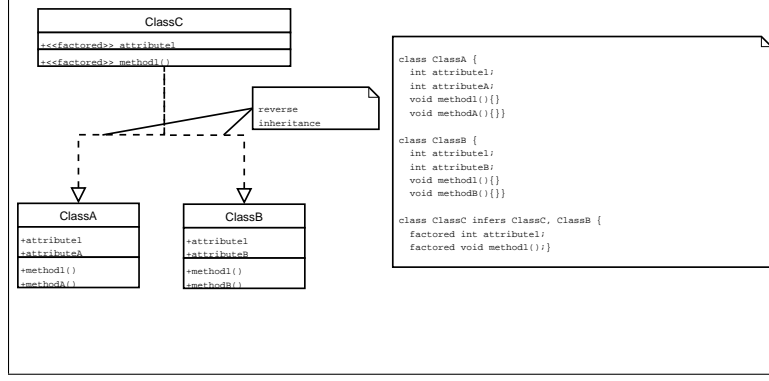
Figure 1. Reverse inheritance factoring mechanism

3. SYNTAX AND IMPLEMENTATION OF FACTORING MECHANISM

In this section we discuss the language syntax and the implementation of the factoring mechanism. In order to show the feasibility of the factoring mechanism we decided to use code transformations to eliminate the reverse inheritance class relationship and to build equivalent hierarchies using just inheritance relationship. We will generate internally equivalent pure Java code and this code does not intend to be shown to the programmer. We propose to analyze examples of the two main situations where reverse inheritance may be involved (single and multiple reverse inheritance). Hierarchy 1 of figure 2 shows an example of single reverse inheritance. The
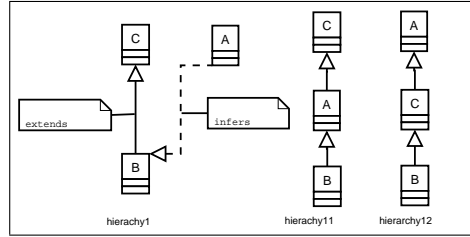
Figure 2. Implementation 1

equivalent *hierarchy11* class diagram can be used when we want to add, to abstract, to redefine, to rename methods in class *B*. The *hierarchy12* from figure 2 can be used when we need to affect in the same way features from class *B* and inherited features from class *C*.

Proposed Java Syntax We propose the following language extension constructions that illustrate single reverse inheritance and fit the context corresponding to the structure of the *hierarchy1* class diagram:

```
class C {}
class B extends C {
  void feature() {/* implementation */}
  void future_factored_feature() {/* implementation */}
  void future_renamed_feature() {/* implementation */}}
class A infers B {
  int new_attribute;
  void new_method(){}
  factored void future_factored_feature();
  void renamed_feature()={void B.future_renamed_feature()};}
```

11

For the abstraction of features the capability to be used is the factoring mechanism. Abstracted features are declared using the *factored* keyword.

Another situation, a more complex one, is presented in hierarchy 2 of figure 3; it deals with multiple reverse inheritance.
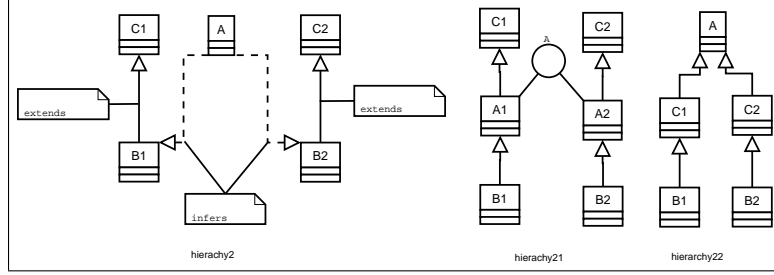
Figure 3. Implementation 2

Like it has been made for single reverse inheritance, we propose two possible implementations depending on the features to be factored. The first implementation solution *hierarchy21* will be used to affect classes *B1* and *B2*. It has two intermediate classes *A1* and *A2* between *C1* and *B1*, respectively *C2* and *B2*. On the other hand an interface *A* is added, which is implemented by the new inserted classes *A1* and *A2*[‡]. Interface *A* will be used to record any possible abstracted features from it's implementing classes *A1* and *A2*. The second implementation solution proposed in *hierarchy22* may be used when we want to affect classes *C1* and *C2*. It has only one superclass A that will include all the factored features and all the new added features. This implementation may be used when we need to factor also inherited features from classes *C1* and *C2*.

4. SIGNATURE MATCHING AND ADAPTATION

In this section we present aspects dealing with signature matching and adaptation in the framework of method factorization. It is necessary to set the rules which define the methods (to be factored) from subclasses that may match the method signature within the superclass and to adapt their signature when it is needed. This may lead to type conversion and to some method renaming. To adapt the signature of methods of subclasses when they are factored enables to make them conform to the signature of the superclass method. This is quite useful because it extends the expressiveness of the factorization mechanism in order to apply it to more methods.

All the entities that contribute to the definition of a signature, are involved in the feature lookup: return type, method name, number of parameter, name, type, position and default value (when available) of each parameter, assertions such as preconditions and postconditions (available in Eiffel)[§].

The work described in [8] involves only signature matching which is based strictly on type analysis. Several cases of possible signature matches are mentioned: exact match, partial relaxed match, transformation relaxed match, combined relaxed match, generic match. Cases of relaxed match, where types are substituted with conforming ones, will be used in our study to reach the goals mentioned earlier.

Potential solutions for name matching, which link concrete methods from subclasses with the correspondent abstract ones in the superclass, are: the use of annotations (a set of meta-information written by the programmer in the source code), information that can be extracted from comments, manual setting of the factored features.

In our approach we address the latter solution and we extend the syntax of Java in order to improve the expressiveness of reverse inheritance class relationship. A sample written in the Java language extension has the following flavour:

[‡]According to the integration of an adding mechanism with reverse inheritance relationship, classes *A1* and *A2* could be used for storing the new added features.

[§]Assertion handling is out of the scope of the paper.

```
01 class Parallelogram {
02   void paint(Canvas c, int x, int y) {/* parallelogram implementation */}}
03 class Ellipse {
04   void update(double x, double y, Canvas canvas) {/* ellipse implementation */}}
05 class Shape infers Parallelogram, Ellipse {
06   factored void paint(Canvas c, int x, int y)= {
07     Parallelogram.paint(Canvas c, int x, int y),
08     Ellipse.update(double x -> x, double y -> y, Canvas canvas -> c)};}
```

Line *05* introduces a new keyword *infers* for expressing the reverse inheritance between class *Shape* and classes *Parallelogram* and *Ellipse*. Between lines *06* and *08*, the common painting method is factored *void paint(Canvas c,int x, int y)*, which corresponds to method *paint(...)* from class *Parallelogram* and *update(...)* from class *Ellipse*. The *factored* keyword is used for marking the factored features in superclass. Also special syntax is used for method and parameter unification: one subclass method has the same name as the factored method *paint*, but the other has a different name *update*; parameter *Canvas canvas* is unified with *Canvas c*, which is not at the same position in class Ellipse.

Discussion About the Implementation of Signature Adaptation The set of transformations that deals with signature adaptation, corresponds to these atomic features: i) method/parameter renaming; ii) parameter addition/removal/reordering; iii) return type of method/parameter type changing.

To decide weather the name chosen for one method of the superclass may be propagated to the name of one subclass according to the example above is not straightforward. In particular, to rename the method in a subclass implies to parse all method calls in order to update it according to the new name. Moreover this may lead to name conflicts with other existing method names.

Furthermore about parameters we may consider renaming, addition and removal. To rename parameters in a method implies changing all the references to these parameters within the same method. Like for method names, name conflicts may arise if members and parameters have now the same name.

To add a new unused parameter will not yield any modification of the method code, but it may interfere with the lookup mechanism and the name of the new parameter could also introduce a conflict with members or other parameters.

To remove an unused parameter implies the identification of the code parts which depend on it, and to evaluate the side effects which are caused by the removal of that code. At first glance, it does not seem quite reasonable.

To reorder parameters within a method is orthogonal according to the code located in the body of that method. But it is obvious that heir or client classes which use the method, will be affected.

If a return type of a method or the type of a parameter is changed then it must conform to the original type, but it is not sufficient because the modification may interfere with the lookup and generate conflicts with existing methods[¶]. About type conformance, the rules proposed for handling primitive types should be the one found in the literature (e.g. in Java *double* can replace *float*). When the type deals with classes then a subtype conforms to the its supertype. But there is also another constraint: clients should use only the feature of the supertype.

Possible Implementation in Java According to the discussion made in previous paragraph, we propose a possible implementation solution which keeps the interface of the class intact and adds delegated methods with different names:

[¶]Moreover in Java the redefinition is non-variant and it is not possible to have two methods with the same name and parameters but with different return types.

```
class Ellipse {
    void update(double x, double y, Canvas canvas) {/* ellipse implementation */}
    void paint(Canvas c, int x, int y) { update(x,y,c); }}
```

In the implementation solution, class *Ellipse* keeps it's *update(...)* method intact and a new method *paint(...)* is added to perform factorization. The new added method *paint(...)* will be used as a delegated method which calls the *update(...)* method using the appropriate order of parameters. Of course it will be necessary to check that there is no name conflict.

Again, code transformation is used only for an implementation purpose. Those we are using, are a set of rules which translates a class hierarchy restructured using reverse inheritance class relationship, into a hierarchy having only normal, direct inheritance. The set of transformations deals with the functionalities discussed in previous paragraph.

5. CONCLUSIONS AND FUTURE WORK

In this paper we studied a possible semantics of the factorization mechanism in the framework of reverse inheritance relationship including signature adaptation and matching. We did not address the impact of reverse inheritance on some Java constructs. For example, we did not point out the semantics of the factorization according to the modifiers of methods or attributes.

Moreover we did not describe neither the semantics nor the implementation when reverse inheritance deals with interfaces or inner classes. Even if the analysis made in this paper is not complete, it suggests that reverse inheritance may be an interesting approach which may improve the reusability and the evolution capabilities of hierarchies of classes. In the near future we aim to finish the definition of the semantics and to validate it by an implementation as a plugin of Eclipse.

Even if it is far to be our first issue, extensions of Java or more generally of any object-oriented language with reverse inheritance may also be used in the context of the reorganization of hierarchy of classes as it has been suggested in 1. In this case it will only play the role of a specification language.

REFERENCES

1. Ciprian-Bogdan Chirila, Pierre Crescenzo, and Philippe Lahire. Towards reengineering: An approach based on reverse inheritance. Application to Java. Research report, Laboratoire Informatique, Signaux et Systemes de Sophia-Antipolis (UNSA / CNRS), France, July 2003.
2. Michel Dao, Marianne Huchard, Therese Libourel, and Cyril Roume. Evaluating and optimizing factorization in inheritance hierarchies. In *Proceedings of the Inheritance Workshop at ECOOP 2002*, Malaga, Spain, June 2002.
3. Peter H. Frohlich. Inheritance decomposed. In *Proceedings of the Inheritance Workshop at ECOOP 2002*, Malaga, Spain, June 2002.
4. Ted Lawson, Christine Hollinshead, and Munib Qutaishat. The potential for reverse type inheritance in Eiffel. In *Technology of Object-Oriented Languages and Systems (TOOLS'94)*, 1994.
5. William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring, 1993.
6. C. H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 407–417. ACM Press, 1989.
7. Markku Sakkinen. Exheritance - Class Generalization Revived. In *Proceedings of the Inheritance Workshop at ECOOP 2002*, Malaga, Spain, June 2002.
8. Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.

14

Mathematical Use Cases lead naturally to non-standard Inheritance Relationships: How to make them accessible in a mainstream language?

Marc Conrad[a], Tim French[a], Carsten Maple[a], Sandra Pott[b]

[a]University of Luton, LU1 3JU, UK
[b]University of York, YO10 5DD, UK

ABSTRACT

Conceptually there is a strong correspondence between Mathematical Reasoning and Object-Oriented techniques. We investigate how the ideas of Method Renaming, Dynamic Inheritance and Interclassing can be used to strengthen this relationship. A discussion is initiated concerning the feasibility of each of these features.

Keywords: Object Oriented Programming, Mathematics, Inheritance, Algebra

1. INTRODUCTION

A strong relationship has already been identified between inheritance relationships and algebraic structuring of "pure" mathematics.[1,2] This relationship is best explored in a highly dynamic object oriented programming language. In practice mathematicians in the course of their undergraduate studies might be exposed only (if at all!) to one of the mainstream languages, such as Java, C++, or perhaps in the near future to C#.

This paper proposes how a mainstream language could be "smoothly" extended to embody new inheritance relationships so as to make them accessible to a mathematical community. We discuss only Java for simplicity, however the proposed additions are, in principle, valid for any mainstream language that only supports inheritance in a manner similar to that of Java, such as C++ or C#.

The aim of the paper is to open a discussion about the feasibility of such extensions. We note that there are basically four possible courses of implementation: 1) Adding the features to Java/C++/C# themselves (that would imply negotiations with the responsible groups/companies); 2) Providing an "add-on" to the mainstream language (as a library or pre-processor); 3) Developing a new language that extends the existing mainstream language; 4) Developing a new language and educating users (universities, mathematics departments and so forth) so as to use this language for mathematical purposes.

2. OBJECT-ORIENTATION AND MATHEMATICS

In the field of Computer Algebra there are already packages that offer support for the object-oriented paradigm. For example, Axiom[3] has type hierarchies ordered in an inheritance-like structure and similarly Mupad[4] explicitly enables the defining of child classes of existing classes as groups, fields, etc.

The author's approach focuses on an object-oriented implementation of mathematical structures in an axiomatic manner.[2,5] The philosophy therein is that postulated properties of a domain are reflected as abstract methods. For example an algebraic ring *has* by definition addition and multiplication. Of course, addition and multiplication are not known *algorithmically* for an arbitrary (unspecified) ring: they cannot be implemented. Therefore the mathematical entity "algebraic ring" is implemented as an abstract base class. This design follows the GoF[6] mediator pattern : The abstract ring class is an abstract *mediator* whilst the elements of the ring are the mediated colleagues.

Email: Marc.Conrad@luton.ac.uk, Tim.French@luton.ac.uk, Carsten.Maple@luton.ac.uk, sp23@york.ac.uk

3. NON-STANDARD INHERITANCE

In this section we address Method Renaming, Dynamic Inheritance, and Interclassing. Each of these three subsections is divided into the presentation of the mathematical Use Cases together with a discussion. For all three features we provide two example Use Cases: An elementary (rather basic) example to introduce and cement the ideas and a more advanced example to demonstrate the power of the method.

3.1. Overriding with Renaming

3.1.1. Mathematical Use Cases

A group is a set with an operation and certain properties. In a concrete situation there is often a standard notation for the group operation. The most familiar are $+$ for addition in an additive group and $*$ or $\times$ for multiplication in a multiplicative group.

Similarly the *composition* of two endomorphisms in the ring of endomorphisms over a vector space becomes matrix *multiplication* in the special case of vector spaces of finite dimension.

3.1.2. Discussion

In these two examples it becomes clear that the renaming of a certain operation after specialization is a familiar task in mathematics. Especially the second example where composition becomes multiplication shows that renaming is able to reflect nontrivial mathematical relationships. This is even more evident in a language that supports operator overloading (as C++): The group operation $a \circ b$ is renamed in a concrete application as either $a \cdot b$ or $a + b$. A well known example can be found in cryptography: A public key/private key algorithm can be formulated for a certain class of abelian, finite groups. The two kinds of groups that are used in practice are $(\mathbf{Z}/n\mathbf{Z})^*$ (a *multiplicative* group) and elliptic curves (*additive* groups). A generic approach[7] dealing with both kinds of groups needs to be supported by a renaming mechanism.

The renaming of an operation after it has been overridden is hardly a new feature in object-oriented contexts. In Eiffel renaming is the preferred method of choice to avoid ambiguity in multiple inheritance relationships.[8] Renaming also exists as a standard feature in Python.[9] Adding this concept to Java would be an easy step to improve the usability of Java within "mathematical context". For example, C# already provides an *overrides* keyword and from here it would be a comparatively small step to extend this syntax by specifying *what* it is that is overridden.

3.2. Dynamic Inheritance

3.2.1. Mathematical Use Cases

Assume we start with an inheritance relationship with *Field* as a child class of *Ring*. For some rings, for example $\mathbf{Z}/n\mathbf{Z}$, it cannot be decided by a compiler a priori if it is a field or not. ($\mathbf{Z}/n\mathbf{Z}$ is a ring if and only if n is a prime number).

In algebraic ring theory we have an even more extended inheritance hierarchy with (for example) *Euclidian Ring*, *Noetherian Ring*, *Principal Ideal Ring* as classes located between *(commutative) Ring* and *Field*. For instance if we restrict consideration to only the class of quadratic orders $\mathbf{Z}[\sqrt{d}]$ with $d \in \mathbf{N}$ we find Euclidian rings and Principal Ideal Rings for various values of d.[10]

3.2.2. Discussion

Dynamic Inheritance is hardly a "new" feature. A C++ implementation (or rather a workaround) is already discussed in.[11] Related to this is the concept of *predicate classes*[12] that is implemented in Cecil.[13] In Self,[14] the object itself can decide on its parent objects thereby giving maximum flexibility. Kniesel[15] proposes a Java extension featuring Dynamic Inheritance. Dynamic Inheritance is also supported in Lava as part of the Darwin project.[16]

The most appropriate approach may well be reclassification as introduced in *Fickle*[17]: An object is related to a *Root Class* (in the Use Cases the class *Ring*). Then it can be reclassified to each child class (called *State*

Classes) of this Root class. A special operator *!!* in *Fickle* reclassifies an object from one State Class to another when both belong to the same Root Class.[17]

The translation of Java into *Fickle* described in[18] may serve as a roadmap for an implementation of reclassification in Java (although it is not straightforward). For a code example that illustrates the proposed syntax of Java reclassification see.[19]

3.3. Interclassing

3.3.1. Mathematical Use Cases

Assume for the moment that Euclid is a contemporary mathematician who has just discovered Euclidian division (also known as division with remainder), and that he wants to add Euclidian division into existing mathematical software that features a ring/field implementation as described in the previous section. The proper place for a Euclidian Ring – a ring with Euclidian division – is between the ring and field. Not every ring is Euclidian and every field is trivially a Euclidian ring.[20]

A typical, non-fictional, example taken from Functional Analysis is that of Triebel-Lizorkin spaces, which were introduced in the 1970's as simultaneous generalizations of a number of well-known classes of function spaces, such as L^p spaces, Hardy spaces, the space of functions of bounded mean oscillation (BMO), Lipschitz spaces and Sobolev spaces.[21]   A Triebel-Lizorkin space is a specialization of a Banach function space. A more recent example is that of so-called real Q-spaces, which are simultaneous generalizations of the space BMO and certain other Banach function spaces.[22]

This "interclassing" in Mathematics is often motivated by the desire to create a unifying framework for several known classes of mathematical objects in a certain context (as in the first of the two examples mentioned above), or to bring existing mathematical techniques to new applications (the second example).

3.3.2. Discussion

Note that the problem of interclassing is substantially different from the problem of run-time reclassification described earlier in section 3.2. Here, we start with a class hierarchy that may be arranged in a package and that may not even contain any source code. We want to extend this class hierarchy by adding a class on a well defined position in an inheritance tree. Even if the source code is available it may not be desirable to change this code, especially if the class library is well established and the addition of the new class has an *experimental* character, or is only relevant for a specialized application area.

Outside of a mathematical context, the idea of *interclassing* is already discussed by Rapicault and Napoli.[23] Crescenzo and Lahire[24] describe an implementation using the OFL model. However, in terms of pragmatic usage OFL is inadequate as it requires *de facto* the learning of OFL as an additional language, namely the understanding of the correct use of hyper-generic parameters. Also, in using hyper-generic parameters, the developer of a library already unnecessarily restricts possible extensions.

A concept developed in LPC[25] called "shadowing" may be a useful technique for the implementation of interclassing. Essentially a shadow is a proxy-object that can be added at run-time and receives all messages determined to the shadowed object (hence "shadowing" the "proxied" object). The concept is evaluated in more detail in two preprints by the authors[19, 26] and has been implemented in Java.[27]

4. CONCLUSION AND PERSPECTIVE

This paper describes work in progress. However a simple (in terms of usability) incorporation of Method Renaming, Dynamic Inheritance, and Interclassing in a mainstream language would radically simplify the implementation of mathematical structures in a wide range of use cases, even in areas that are currently merely considered as practically not accessible by programming (such as Functional Analysis).

The paper was motivated by disappointment with the traditional way of implementing "mathematics" within mainstream Computer Algebra Systems and experiments using the com.perisic.ring Java package.[5]   However, it seems that the question of "implementing" mathematical structures in an object oriented context is strongly linked to (and may be dominated by) the issue of how best to represent these structures.

Moreover, it may be fruitful to discuss the problems of Method Renaming, Dynamic Inheritance and Interclassing independently from any implementation language in the context of the UML. Having appreciated the usefulness of a UML representation of mathematical structures, the use cases provided in this paper may lead to future examination of suitable extensions to UML, for instance how Reclassification should be modelled in a Sequence Diagram etc.

REFERENCES

1. M. Conrad, *Abstract Classes - pure computer science meets pure mathematics*, Seminar talk, York 2003, http://ring.perisic.com/info/york2003.

2. M. Conrad, T. French, "Exploring the synergies between the Object-Oriented paradigm and Mathematics: a Java led approach," to appear in *Int. J. Math. Educ. Sci. Technol.*

3. Tim Daly et. al. *Axiom Computer Algebra System*, http://savannah.nongnu.org/projects/axiom.

4. The MuPAD Research Group. *MuPAD – The Open Computer Algebra System,* http://www.mupad.de.

5. M. Conrad. *com.perisic.ring – A Java package for multivariate polynomials*, http://ring.perisic.com.

6. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns,* Addison-Wesley, 1995.

7. C. Maple, M. Conrad, T. French, "A Novel Flexible Approach to Document Encryption Using a MathML Extension to the W3C XML Digital Certificate Standard," in *Proceedings of the IADIS International Conference on e-Society* (2003)

8. Betrand Meyer, "Overloading vs. Object Methodology," *Journal of Object-Oriented Programming*, October/November 2001.

9. Jeremy Hylton, *Introduction to Object-Oriented Programming in Python (Outline)*, http://www.python.org/˜jeremy/tutorial/outline.html, Januar 2000.

10. Kenneth Ireland, Michael Rosen, *A Classical Introduction to Modern Number Theory, 2nd ed.* Springer-Verlag, New York, 1995

11. James Coplien, *Advanced C++ programming styles and idioms*, Addison-Wesley 1992.

12. C. Chambers, "Predicate classes," in: *Proceedings of the ECOOP'93*, volume 707 of *Lecture Notes in Computer Science*, pages 268–296, Kaiserslautern, Germany, July 1993.

13. C. Chambers, *The Cecil Language: Specification & Rationale*, avialable at: http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html.

14. The Self Group, *Self*, http://research.sun.com/research/self

15. Günter Kniesel, *Darwin & Lava - Object-based Dynamic Inheritance ... in Java*, Poster presentation at ECOOP 2002.

16. *The Darwin Project*, http://javalab.iai.uni-bonn.de/research/darwin.

17. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini and P. Giannini, "Fickle: Dynamic object reclassification," in: *ECOOP'01*, LNCS **2072** (2001), pp. 130–149.

18. D. Acnona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, E. Zucca, "A type preserving translation of Fickle into Java," in: *Electronic Notes in Theoretical Computer Science* 62 (2001). Available at: http://www.elsevier.nl/locate/entcs/volume62.html

19. M. Conrad, T. French, C. Maple, S. Pott, *Approaching Inheritance from a "Natural" Mathematical Perspective and from a Java driven viewpoint: a Comparative Review*, Preprint available from: http://ring.perisic.com.

20. Serge Lang, *Algebra*, third ed., Addison-Wesley, 1993.

21. H. Triebel, "Theory of Function Spaces," *Monographs in Mathematics*, vol 78, Birkhäuser Verlag Basel, 1983

22. M. Essén, S Janson, L. Peng and J. Xiao, "Q-spaces of several real variables," *Indiana University Mathematics Journal*, vol 49, no 2(2000), 575 – 615

23. P. Rapicault, A. Napoli, "Evolution d'une hirarchie de classes par interclassement," in: *LMO'2001, Hermes Sc. Pub. "L'objet"*, vol. 7 - no. 1–2/2001

24. Pierre Crescenzo, Philippe Lahire, "Using Both Specialisation and Generalisation in a Programming Language: Why and How?" In: *Advances in Object-Oriented Information Sytems*, OOIS 2002 Workshops, Montpellier, pages 64–73, September 2002.

25. Ronny Wikh, *LPC*, available at: http://genesis.cs.chalmers.se/coding/lpcdoc/lpc.html (last update 2003)

26. M. Conrad, T. French, C. Maple, "Object Shadowing - a Key Concept for a Modern Programming Language," Submission to the *2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity* (Workshop 5 of ECOOP 2004).

27. M. Conrad. *The com.perisic.shadow package*, http://perisic.com/shadow.

Proposals for Multiple to Single Inheritance Transformation

Michel Dao[a], Marianne Huchard[b], Thérèse Libourel[b], Anne Pons[c], Jean Villerd[b]

[a]France Télécom R&D DTL/TAL
38-40 rue du général Leclerc, 92794 Issy Moulineaux Cedex 9, France
[b]LIRMM – CNRS et Université Montpellier II – UM 5506
161 rue Ada, 34392 Montpellier Cedex 5, France
[c]Dept Informatique – Université du Québec à Montréal
C.P. 8888 – Succursale centre ville, Montréal Québec H3C 3P8, Canada

ABSTRACT

We present here some thoughts and ongoing work regarding transformations of multiple inheritance hierarchies into single inheritance hierarchies. We follow an approach that tries to categorize multiple inheritance situations according to a semantic point of view. Different situations should be captured through diagrammatic UML annotations that would allow to detect a given situation and hence apply the appropriate transformation, automatically if possible.

Keywords: multiple inheritance, UML, model transformation

1. INTRODUCTION

Multiple inheritance (MI) vs. single inheritance (SI) was the subject of numerous passionate discussions during the era of the emergence of object-oriented programming languages. Those discussions have more or less come to an end and we are left with: not so badly implemented MI languages (e.g. Eiffel and CLOS) that are scarcely used, widespread not so well implemented MI languages (e.g. C++) or the flagship language Java (but for how long?) with SI (and MI for interfaces).

Lately, a shift in software development process has given an increased importance to modelling, specially through the widespread use of UML.[1] Of course UML proposes MI: as opposed to programming languages, there are no conflicts to be resolved at compile time when using MI in UML and those who believe, as we do, that MI can be a good means of modelling existing entities can use it without caution. Furthermore, UML proposes annotations allowing inheritance links to convey special meanings.

More recently, Model Driven Architecture (MDA)[2] has proposed a framework in order to formalize the extensive use of models during the software development process. MDA fosters the use of different models throughout the process, the models of one phase of development being derived from the models of a previous phase. More specifically, a Platform Independent Model (PIM) may be used to generate a Platform Specific Model (PSM). For instance, a UML design level static model may be used to generate source code in a given programming language. Our proposition fits into this precise scheme: how can we automate the transformation of a MI UML class diagram into a SI class diagram, hence allowing straightforward transformation into SI programming language?

There exist several works on the subject of MI vs. SI.[3–5] In a previous project in which we participated, two approaches have been considered that may eventually be combined. The first one may be described as "combinatorial" and consists in defining a strategy to remove inheritance links so as to minimize the number of properties (attributes and methods) that it is necessary to duplicate. Such a strategy can be based on a set of metrics that allow to measure *a priori* the impact of the deletion of an inheritance link. A first work following this approach has been realized[6] that yielded interesting results but needs to be refined and completed in order to be fully usable.

Another approach, which is the subject of this article, may be described as "semantic". The idea is to consider that MI may appear in several typical situations that correspond to different semantics and that for each situation there may exist several possible transformations into SI. The problem of MI to SI transformation may therefore be decomposed as follows:

- elaborate a list of the different situations of MI and of the corresponding possible transformations into SI;

- be able to find occurences of those different situations in a class hierarchy;

- be able to apply the pertinent transformation.

The structure of inheritance is clearly not sufficient to determine a situation of MI. UML standard proposes some annotations of inheritance and we believe that those annotations may help in spotting specific inheritance situations but they are limited and do not allow to capture all situations. In a previous article,[7] we have proposed some extensions to those annotations in order to enhance the semantic expressiveness of inheritance links in UML class diagrams. Furthermore, some other informations (size of classes, size of generalization set, need of symmetry, number of inherited methods, etc.) may help to determine the best transformation to be applied.

We first present the annotations that may be used to convey semantic inheritance information in UML class diagrams. This is followed by a proposition of a set of MI to SI transformations that we have gathered in existing work (and have partly adapted). Then we propose a tentative list of typical MI situations associated with one or more pertinent transformations. We conclude by discussing our approach and its perspectives.

2. MULTIPLE INHERITANCE ANNOTATIONS

A first type of UML annotation[1,8] is the *discriminator* that allows one to group subclasses into clusters that correspond to a semantic category. For instance, in Figure 1*, class `Employee` is specialized according to two criteria, `:status` (related to the salary payment), and `:pension` (vested vs unvested). Discriminators involve a partition of the specialization links coming to a parent class: in our example this partition has two elements, the set of the links labelled with the discriminator `:status` from one side, the set of the links labelled with the discriminator `:pension` from the other side.
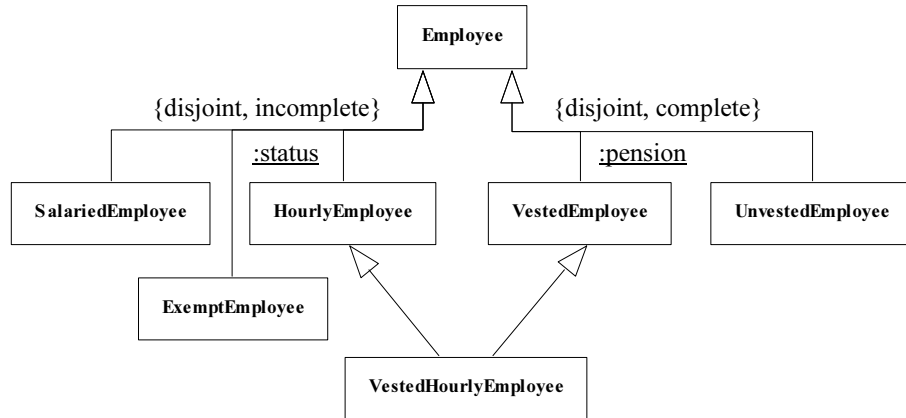
Figure 1. Exemple of annotated multiple inheritance

A second kind of annotation existing in the UML notation appears as constraints about the extension (instance set) of a class C and of its subclasses. We denote by E the set of the direct subclasses which are gathered by such an annotation. Four constraints are predefined:

overlapping an instance of C can simultaneously be instance of several classes of E;

disjoint an instance of C is instance of at most one class of E;

complete elements of E are origins of links annotated by a same discriminator; any instance of C is instance of one of the elements of E;

---

*This example is borrowed from.[9]

22

incomplete the classes of E are origins of links annotated by a same discriminator; an instance of C is not necessarily instance of one of the classes of E.

We have proposed to extend this set of annotations with the following[7]:

alternative the characteristics of the super classes are used alternatively as in the case of an amphibian vehicle;

concurrent (special case of overlapping for roles and states): father/husband;

successive (special case of disjoint with a temporal scheduling): chrysalis/caterpillar/butterfly, child/teenager/adult;

exclusive (special case of disjoint for roles and states): empty/full for a stack, married/single for a person;

repeated similar to repeated inheritance in C++: a property may be inherited along several different paths from the same indirect super class;

combined when a class is directly specialized according to several criteria denoted by discriminators, an instance may be constrained to belong to at least one class of each discriminator;

disjoint partition conversely, an instance may be constrained to belong to only one discriminator;

implementation in numerous examples of multiple inheritance in the programming area, a class ends up deriving from superclasses that it specializes for implementation needs.

3. INHERITANCE TRANSFORMATIONS

Figure 2 shows an initial MI situation and five possible transformations into SI.

Transformation 1 – Duplication The first transformation that may be applied is to remove one of the inheritance links and to duplicate in the subclass all the properties that were inherited through this link. The advantage of this solution is simplicity but duplication of code is always a bad thing regarding reuse and maintenance. The choice of the inheritance link to cut could be based on the number of properties that must be duplicated: the less, the better.

Transformation 2 – Nested generalizations This transformation consists in cloning one set of classes corresponding to a discriminator (here :discrCD) into subclasses of each class corresponding to the other discriminator. This transformation may only be useful when there are few classes under the chosen discriminator and it may be difficult to choose the discriminator to clone. Furthermore, the naming conflicts produced by the new classes must be resolved but polymorphism is kept.

Transformation 3 – Direct link This transformation can be seen as a double duplication: both inheritance links are cut and the properties from both superclasses (B and C) are duplicated into class E. This transformation involves more duplication than the duplication involved in transformation 1 but allows to preserve the symmetry of the class hierarchy if this is relevant.

Transformation 4 – Role aggregation Another solution is to transform one of the inheritance links into an aggregation[†] link. Polymorphism is replaced by a delegation mechanism at the expense of the creation of a new class A/CD and of code rewriting. In our example each method or accessor of class C should be replaced by one with the same name in class A performing a call to the right method or accessor in class C. The choice of the inheritance link to be replaced by delegation could be based on the amount of properties to be redefined or one could choose the class belonging to a discriminator that is complete because such a replacement would be done once and for all.

[†]In fact, that might be a composition link.
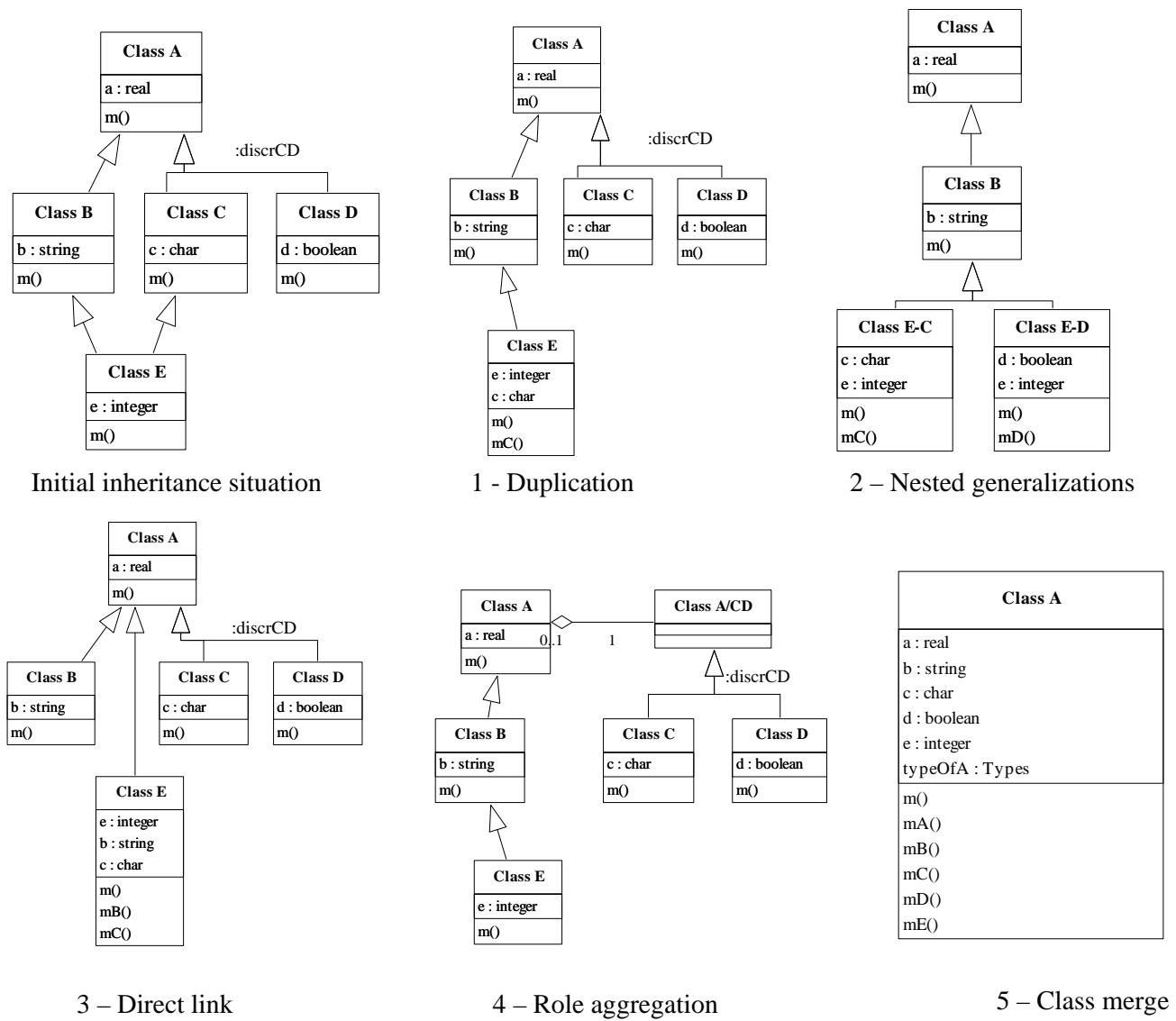
Figure 2. Propositions of inheritance transformations

Transformation 5 – Class merge   This transformation merges `A` and all its subclasses into one unique class. Conflicting properties must be renamed and a dispatch mechanism must be implemented that uses a typing attribute indicating the type of instances of this unique class.

Transformation – Interfaces   Another transformation not depicted in the figure consists in defining a MI interface hierarchy corresponding to the MI class hierarchy and to establish implementation links between the class hierarchy and the implementation hierarchy.

Our ongoing work is to define a mapping between a given MI situation and one (or several) pertinent transformations. Table 1 lists a first attempt of such a mapping. This is clearly an incomplete and debatable table that needs completion, refinement and discussion. Let us discuss a couple of our proposals.

First, the *complete* annotation implies that no other classes should be added to the cluster of classes gathered under this annotation, typically a discriminator. Therefore, in the case where there are only few classes involved, one multiple inheritance cluster could be transformed using the *nested generalization* transformation. In this

24

case, this would not lead to a too complex diagram with a great number of classes that are not all useful. Another case is when there is only one inheritance link that originates from a *complete* annotation, one could use the *role aggregation* transformation with this inheritance link being replaced by the aggregation link.

The *combined* annotation stipulates that a subclass inherits from at least one class from each discriminator. In the case of the *alternative* sub case, this argues in favor of the *direct link* transformation where all superclasses are treated equally.

The *implementation* annotation can be viewed as a conceptual model of the interfaces of Java (*able: cloneable, serializable, etc.) and therefore the most natural transformation consists in using interfaces to represent multiple inheritance in that case.

Situation	Semantic subsituation	Transformation	Comment
overlapping	concurrent	role aggregation	
disjoint	successive exclusive	role aggregation	
complete		nested generalization role aggregation	few classes under the chosen discriminator and at most two discriminators complete = will not evolve
combined	alternative	role aggregation nested generalizations direct link	few classes under the chosen discriminator
repeated		role aggregation	
implementation		interfaces	

Table 1. MI situations and transformations

4. DISCUSSION AND PERSPECTIVES

We have presented here our ongoing work on MI to SI transformation based on semantic annotations of UML class diagrams. We have so far enriched UML annotations with some new ones and determined a set of transformations. We are currently studying the mapping between a given situation of inheritance (UML extended annotations and other criteria) and the possible transformations that may be applied.

It is obvious that such transformations should be applied to a class hierarchy as automatically as possible. We have realized a limited implementation of two of the transformations listed in Section 3 in UML CASE tool Objecteering[‡] using its proprietary object-oriented language J. We are wondering if this type of procedural implementation is best suited for our purposes. As inheritance transformation may be seen as model transformation, we are considering the use of a model transformation language (such as those for which OMG is requiring for proposals) to express both the research of MI inheritance situations and their transformations into SI.

We believe that the work we have presented here may be an incentive for the following discussion topics:

- can we reconcile multiple and single inheritance by allowing the latter to be (partially) automatically obtained from the former?

- does this type of transformations fit into the MDA approach?

- to which extent can we classify multiple inheritance into well defined semantic categories?

- to which extent can we capture those semantic categories in UML annotations?

[‡] `www.objecteering.com`

REFERENCES

1. U2 Partners, *Unified Modeling Language: Superstructure, version 2.0, 3rd Revised submission to OMG RFP ad/00-09-02, http://www.omg.org/cgi-bin/doc?ad/20 03-04-01*, april 2003.

2. Object Management Group, *MDA-Guide, V1.0.1, omg/03-06-01*, june 2003.

3. K. Thirunarayan, G. Kniesel, and H. Hampapuram, "Simulating Multiple Inheritance and Generics in Java," *Computer Languages* **25**(4), pp. 189–210, 1999.

4. M. Malak, "Simulating Multiple Inheritance," *Journal of Object-Oriented Programming* , pp. 3–5, april 2001.

5. Y. Crespo, J.-M. Marquès, and J. Rodriguez, "On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies," in *Proceedings of the Inheritance Workshop at ECOOP 2002*, Black, Ernst, Grogono, and Sakkinen, eds., pp. 30–37, 2002.

6. C. Roume, "Going from Multiple to Single Inheritance with Metrics," in *Proceedings of the sixth ECOOP workshop on Quantitative Approaches in Object Oriented Software Engineering (QAOOSE 2002)*, F. Brito e Abreu, M. Piattini, G. Poels, and H. Sahraoui, eds., pp. 30–37, 2002.

7. M. Dao, M. Huchard, T. Libourel, and A. Pons, "Extending the Notation for Specialization/Generalization," in *Proceedings of MASPEGHI'03, ISBN 2-89522-035-2*, pp. 61–67, (CRIM, Université de Montréal), 2003.

8. Rational Software Corporation, *UML v 1.3, Notation Guide*, version 1.3 ed., june 1999.

9. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object Oriented Modeling and Design*, Prentice Hall Inc. Englewood Cliffs, 1991. pages 15–84.

The expression problem, Scandinavian style

Erik Ernst

Dept. of Computer Science, University of Aarhus, Denmark
`eernst@daimi.au.dk`

ABSTRACT

This paper explains how higher-order hierarchies can be used to handle the expression problem. The expression problem is concerned with extending both the set of data structures and the set of operations of a given abstract data type. A typical object-oriented design supports extending the set of data structures, and a typical functional design supports extending the set of operations, but it is hard to support both in a smooth manner. Higher-order hierarchies is a feature of the highly unified, mixin-based, extension-oriented kind of inheritance which is available in the language `gbeta`, which is itself a language that was created by generalizing the language BETA.

Keywords: Expression problem, class composition, gbeta

1. INTRODUCTION

The expression problem has been defined as follows by Torgersen[1]: "Can your application be structured in such a way that both the data model and the set of virtual operations over it can be extended without the need to modify existing code, without the need for code repetition and without runtime type errors."

This paper presents a new approach to the expression problem, based on the support for higher-order hierarchies which is a feature of the inheritance mechanism in the language `gbeta`. It is very straightforward for programmers to express the desired extensions in this style, and moreover different extensions are highly composable.

The expression problem is about two-dimensional extensions: if an abstract datatype is modeled in an object-oriented style by means of classes whose methods are the operations on the datatype, then it is easy to extend the set of variants by writing another class. If the abstract datatype is modeled in the style of an SML `datatype` with a set of pattern matching functions as the operations, then it is easy to extend the set of operations by writing yet another pattern matching function. In both cases it is much harder to perform the *other* extension, i.e., to add a new operation in the object-oriented style or to add a new data structure in the functional style, in both cases because it is necessary to make changes in many existing entities rather than just writing one new entity. Space constraints imply that the traditional forms of the expression problem cannot be presented by detailed code examples here, but this has been done several times before.[1,2] Instead, we will proceed to describe and discuss the new solution.

Section 2 presents the running example in terms of which our solution to the expression problem is exposed, and gives the code for the initial design. This design is extended in Sect. 3 with a new operation, and in Sect. 4 with a new kind of data. Next, Sect. 5 shows how the two extensions may be composed and glued together. Finally, Sect. 6 briefly discusses related work and Sect. 7 concludes.

2. THE EXAMPLE PROBLEM

Following the tradition in relation to the expression problem, we will focus on a compiler-oriented problem, namely that of representing abstract syntax trees for a tiny language. The desired extensions will then correspond to adding a new operation on the abstract syntax trees, and adding a new kind of nodes in the trees. Here is the source code for the initial design:

```
class Lang {
  virtual class Exp {
    String toString() {}
  }
  virtual class Lit extends Exp {
    int value;
    Lit(int value) { this.value=value; }
    String toString() { return value; }
  }
  virtual class Add extends Exp {
    Exp left,right;
    Add(Exp left, Exp right) {
      this.left=left; this.right=right;
    }
    String toString() {
      return left.toString()+"+"+right.toString();
    }
  }
}
```
Ex. 1

We use a syntactic style which is close to the Java programming language,[3] but which is in fact just a modified surface syntax for the language gbeta. In particular, class attributes may be virtual, which means that they may be redefined (more precisely: further constrained) in subclasses of the enclosing class, and this is what we will exploit in order to create the desired extensions later. Note that gbeta supports inheritance between virtual classes; in BETA such an inheritance relation is not supported, but gbeta supports it as a result of a deep generalization of the underlying concepts and mechanisms. This is the basis for higher-order hierarchies, which is described in detail elsewhere.[4]

The class Lang contains three classes Exp, Lit, and Add, the latter two being subclasses of the first one. The class Exp represents abstract syntax trees for expressions in the tiny language we are dealing with, and the two other classes are the only possible forms of expressions, namely integer literals, Lit, and addition expressions, Add. The only operation available in this basic version is toString.

3. ADDING A NEW OPERATION

We can extend the class family with a new operation in the following way (note that we need not edit the base family):

```
class LangEval extends Lang {
  refine class Exp {
    int eval() {}
  }
  refine class Lit {
    int eval { return value; }
  }
  refine class Add {
    int eval { return left.eval()+right.eval(); }
  }
}
```
Ex. 2

The effect of this is that we create a derived class family (a new version of Exp, Lit, and Add), each of them created by extending the version in Lang with the new eval method. The keyword refine is used to specialize an inherited virtual class attribute, and the semantics is that the virtual class attribute is constrained to be a subclass of the new declaration. In other words, no matter what class Exp denotes in Lang, it will denote that same class extended with the eval method in LangEval. Note that Add was declared to be a subclass of Exp in Lang; such a subclass relation is maintained even when virtual classes are refined, and this means that Add in

28

`LangEval` is a subclass of the *current* value of `Exp`, which is then extended with a particular implementation of the `eval` method.

Note that all we had to do in order to extend the entire family with a new operation was to declare which classes should be extended with the new method (`refine..`), and then declare the method. If `Lit` and `Add` could have used an implementation of eval written in `Exp` then we could have written just that single new method in a refinement of `Exp`, and all subclasses (in this case: `Lit` and `Add`) would have inherited the new method without any need to mention them explicitly. This is again because the declared inheritance relations are automatically maintained.

4. ADDING A NEW DATA STRUCTURE

Extending the class family with a new member is also easy:

```
class LangNeg extends Lang {
  virtual class Neg extends Exp {
    Neg(Exp exp) { this.exp=exp; }
    String toString() { return "-(" + exp.toString() + ")"; }
    Exp exp;
  }
}
```
Ex. 3

Here we create a new class family whose members have the same structure as in `Lang` (because there are no `refine` declarations), but a new family member is added, namely `Neg` which represents the unary negation operator. As before, there is no need to edit `Lang` in order to create this extension of it, and the new extended version of `Lang` is created simply by describing the delta—in this case the class to add to the family. Note, however, that the new family member is declared to be a subclass of `Exp`. This means that extensions to `Exp` will also added to `Neg`, as we shall see in the next section.

5. COMPOSING BOTH EXTENSIONS

It is possible to use the two extensions together, by composing the two class families created in Sect. 3 and 4. In gbeta, class composition is supported by means of the '`&`' operator, but since this character is already used for other purposes in Java syntax we will use the (non-ASCII) symbol $\oplus$ to play this role. The class families may then be combined in the following manner:

```
class LangNegEval extends LangEval ⊕ LangNeg {
  refine class Neg {
    int eval() { return -exp.eval() }
  }
}
```
Ex. 4

It is trivial to compose the two class families, producing a new family which contains the base material from `Lang` as well as the added `Neg` class from `LangNeg` and the added `eval` method from `LangEval`. To do this, we can just use `LangEval`$\oplus$`LangNeg`. However, we need to add a little bit of glue code to this combination, because `LangEval` does not know about the class `Neg` and `LangNeg` does not know about the method `eval`, and the result is that the implementation of `eval` for the class `Neg` is non-existent. This might be fine since `Neg` does in fact inherit `eval` from `Exp` in context of the combined family, but the implementation in `eval` in `Exp` is not suitable for `Neg`, so we have to add an implementation of `eval` specifically for `Neg`. This extension is achieved by the body of class `LangNegEval` above.

Note that it is not a problem with the expressive power of the language that forces us to write this glue code, it is a problem which is inherent in the combination of independent extensions. We could never expect the independently added method and the independently added class to match up in such a way that the combination of the extensions would know how to implement the new method for the new class—this is inherently an application domain dependent problem, which must be solved by a programmer who writes the missing method.

6. RELATED WORK

Krishnamurthi et al.[2] describe an extension of the visitor pattern with factory methods is used to ensure new datatypes can be added while maintaining consistency. Adding new operations is easy when using the visitor pattern, so this establishes a two-dimensional extensibility. However, it leads to significantly more complex programs than what we have shown in this paper, it does not support smooth composition of independent extensions, it is not well-integrated in the type system (it uses explicit type casts), and it does not support type-safe polymorphic usage of complete families of classes, also known as family polymorphism.

Torgersen[1] describes a number of visitor based approaches, with a similar level of complexity as in the previous approach and also without family polymorphism or extension composition, but it removes the need for dynamic casts. Moreover, in this case there is a feature which is not available in the approach we have shown: it is, under certain circumstances, possible to mix objects belonging to different families. E.g., an expression may contain nodes from the basic family (without negation), and this expression could then be used as the subtree of a `Neg` node.

Zenger and Odersky[5] describe how SCALA is used to express two different families of solutions to the expression problem which are capable of combining independently added extensions. Two extensions adding datastructures are combined, and two extensions adding operations are combined—and it is not clear whether two extensions can be combined (and glued) if one of them adds a datastructure and the other one adds an operation. SCALA uses some constructs similar to virtual types (it supports abstract type members in objects), and it is more theoretically well-analyzed but a little less expressive than `gbeta`. In particular, the example programs are more complex than the ones we have shown here, because SCALA does not support propagating combination and consequently the combination of nested type members must be spelled out manually.

7. CONCLUSION

We have presented an approach to the expression problem based on higher-order hierarchies. Using this approach, the expression problem becomes a small matter of writing the classes and/or methods which need to be added to a given class family, and it is even possible to combine independent extensions and add the missing glue. We believe that this is the smoothest known type-safe approach to the expression problem.

REFERENCES

1. M. Torgersen, "The expression problem revisited – four new solutions using generics," in *Proceedings ECOOP'04*, M. Odersky, ed., *LNCS ?*, pp. ?–?, Springer-Verlag, (Oslo, Norway), 2004. To appear.
2. S. Krishnamurthi, M. Felleisen, and D. P. Friedman, "Synthesizing ojbect-oriented and functional design to promote re-use," in *Proceedings ECOOP'98*, E. Jul, ed., *LNCS* **1445**, pp. 91–113, Springer-Verlag, (Brussels, Belgium), July 1998.
3. B. Joy, G. Steele, J. Gosling, and G. Bracha, *Java(TM) Language Specification (2nd Edition)*, Addison-Wesley Publishing Company, 2000.
4. E. Ernst, "Higher-order hierarchies," in *Proceedings ECOOP 2003*, L. Cardelli, ed., *LNCS 2743*, pp. 303–329, Springer-Verlag, (Heidelberg, Germany), July 2003.
5. M. Zenger and M. Odersky, "Independently extensible solutions to the expression problem," Tech. Rep. IC/2004/33, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2004.

The Logic of Inheritance

DeLesley Hutchins
CISA, Department of Informatics,
University of Edinburgh
d.s.hutchins@sms.ed.ac.uk

ABSTRACT

Although inheritance is the characteristic feature of object-oriented programming, it has proved difficult to formally define. There is still a great deal of debate over the benefits of single versus multiple inheritance, subtyping versus subclassing, and inheritance of interface versus implementation. I argue that inheritance is fundamentally concerned with the categorization of objects, and that OO languages should thus be founded upon a formalism that supports categorical reasoning. Description logics, which were developed by the AI community for knowledge representation, provide just such a formalism. I provide a brief introduction to description logics, and then describe how they can be extended to create a formal object calculus. The resulting calculus seems to resolve many of the problems that plague other models of inheritance.

Keywords: description logics, feature composition, mixins, mixin layers, multiple inheritance, prototypes, type intersection

1. INTRODUCTION

The object-oriented paradigm suffers from a critical ailment. Inheritance is the main feature that distinguishes OOP from other programming paradigms, and yet there are as many variations of "inheritance" as there are OO programming languages. The greatest strength of inheritance is also its greatest weakness; it is intuitive and easy to understand in principle, but our natural intuitions have proved difficult to satisfactorily define in any formal way.

Part of the problem is that most discussions of inheritance have focused on mechanism, rather than meaning. Inheritance in mainstream languages is closely tied to implementation concerns, specifically the use of virtual method tables. In both C++ and Java, only methods can be overridden during inheritance, because only method pointers can be stored in a lookup table. Although Beta introduced virtual types and classes over a decade ago, they are still not supported by mainstream languages, and that means that inheritance in such languages is fundamentally limited.[1] It is not possible in Java, for instance, to create generic classes, specialize groups of mutually recursive classes, or encapsulate cross-cutting concerns using single inheritance alone. A number of proposals for additional constructs — such as generics,[2] aspects,[3] mixins,[4] and feature composition[5] — have been proposed and implemented in recent years in part as a way of getting around the limitations of single-class method-based inheritance.

This paper is a step toward a more expressive and logical model of inheritance. I will start by defining the *meaning* of inheritance as a subset of first-order logic, and move from there to defining an appropriate *mechanism* for inheritance as a calculus of classes and objects. I conclude by briefly comparing this mechanism with other languages and techniques.

2. DESCRIPTION LOGICS

Description logics (DLs) are fragments of first-order logic that have been developed by the AI community for use in knowledge representation systems.[6] They evolved out of earlier work on semantic networks and frame-based systems, and are currently one of the main logics used to build ontologies for the semantic web. DLs are particularly well-suited to reasoning about categories, which makes them applicable to OO languages.

A description logic models the world in terms of *concepts* and *roles*. A concept is a unary predicate (e.g. $p(X)$) which specifies a property that a given object X may have. A role is a binary predicate (e.g. $p(X,Y)$) which specifies a relationship that may hold between a pair of objects. In a description logic, these predicates

are usually interpreted as sets. We start off with a non-empty set Δ^I which is the *domain of interpretation* – the set of all possible objects. Given a concept C, we can then define C^I, the subset of Δ^I which contains only those objects that have the properties defined by C. A concept thus acts essentially like a type or a class. Similarly, a role R has an interpretation R^I, which is a mathematical relation between objects in Δ^I.

The syntax of DLs may seem somewhat strange to the uninitiated, because predicate variables are not explicit in the syntax. That is, instead of writing $c(X) :- d(X) \wedge e(X)$, we write $C \equiv D \sqcap E$. The use of the "$\sqcap$" symbol rather than "$\wedge$" reflects the fact that it is easiest to think of concepts as types or sets rather than predicates.

One of the big advantages of DLs is that they comprise a family of logics rather than a single logic, so they can be tailored to meet the needs of a particular system. The language I present here is based on the logic $\mu\mathcal{FL}^-(\subseteq, \circ)$, which is described by the following grammar.[6] The semantics of the logic are given in set notation.

$$
\begin{array}{lll}
C, D & ::= & \text{Concepts} \\
\quad \top & & \Delta^I \text{ (set of all objects)} \\
\quad \bot & & \emptyset \text{ (empty set)} \\
\quad A & & \text{Atomic concept} \\
\quad X & & \text{concept self-label} \\
\quad C \sqcap D & & C^I \cap D^I \text{ (concept intersection)} \\
\quad \mu X.\, C & & \text{fixpoint operator} \\
\quad \forall R.\, C & & \{a \in \Delta^I \mid \forall b.\ (a,b) \in R^I \text{ implies } b \in C^I\} \\
\quad \exists R & & \{a \in \Delta^I \mid \exists b.\ (a,b) \in R^I\} \\
\quad R \subseteq S & & \{a \in \Delta^I \mid \forall b.\ (a,b) \in R^I \text{ implies } (a,b) \in S^I\} \\
& & \\
R, S & ::= & \text{Roles/Relations} \\
\quad P & & \text{Atomic role} \\
\quad R \circ S & & \{(a,c) \in \Delta^I \times \Delta^I \mid \exists b.(a,b) \in R^I \wedge (b,c) \in S^I\}
\end{array}
$$

A knowledge base built using description logics starts out by defining several *atomic* concepts and roles which act as built-in types. More complex concepts are then constructed by describing their relations to simpler ones. For example, if Human and Female are atomic concepts, we can write:

Parent	$\equiv$	Human $\sqcap$ $\exists$child $\sqcap$ $\forall$child.Human	A parent is a human with at least one human child.
Mother	$\equiv$	Parent $\sqcap$ Female	A mother is a female parent.
Spinster	$\equiv$	Human $\sqcap$ Female $\sqcap$ $\forall$child.$\bot$	A spinster is a woman with no children.

Description logics make heavy use of the "$\sqcap$" type-intersection operator. Each part of a concept is essentially a very simple class. The concept $\exists$child, for instance, is the class of objects which have at least one child-link to another object. The concept $\forall$child.Human is the class of objects whose child-links, should they exist, all point to humans. The intersection operator then glues a number of such mini-classes together into a more complex class.

Here's another example, this time describing data structures:

Tree	$\equiv$	μX.$\forall$branch.X	The branches of a tree have the same type as the root.
IntNode	$\equiv$	$\exists$item $\sqcap$ $\forall$item.Integer	An integer node has at least one integer item.
IntTree	$\equiv$	Tree $\sqcap$ IntNode	A tree whose nodes are are all integer nodes.

The fixpoint operator μX allows a concept to refer to itself, and is equivalent to the This or MyType construct found in advanced type systems.[7,8] By using the variable X to range over the type of nodes in the tree, we ensure that all of the nodes in an IntTree have an item, not just the root node. In other words, the intersection of Tree with IntNode *binds* X to Tree $\sqcap$ IntNode. This binding action is the key to extending DLs so that they can handle OO programming.

The $\subseteq$ operator is used for defining *role-value maps*, which declare relationships between different attributes of the same object. For example, the concept "humans who love their neighbors" can be written as Human $\sqcap$ neighbors $\subseteq$ loves.

In addition to concept constructors, there is also one role constructor: the *role chain*. The "grandchild" relation can be defined as child ∘ child, while "friends of my children" can be written as child ∘ friend.

2.1. Comparison with OO programming

Although the syntax is quite different, description logics bear a remarkable similarity to the constructs in object-oriented programming languages. As I mentioned before, concepts in a DL are essentially the same as classes in an OO language. Roles, in turn, are similar to fields or slots. A concept is basically a data structure with links that point to other concepts.

There are two main difference between DLs and OO languages. The first difference is that roles in a DL are not necessarily functional relations. The definition of Tree above does not say that trees only have one branch. Trees may have many branches or none at all; the definition of Tree merely places a type bound on any branches that exist.

The second major difference is the fact that "inheritance" in a DL is implemented in terms of concept intersection. There is no such thing as "single inheritance", because it makes no sense to take the intersection of just one concept. "Multiple inheritance" is required merely to create a class with multiple slots. Description logics break classes down into very simple pieces which can be easily recombined, and therein lies their utility.

3. THE SYM CALCULUS

As originally formulated, description logics are really just type systems, because concepts are types. DLs provide a calculus for constructing types, and algorithms exist for determining subsumption (subtyping), satisfiability (type validity), and finding out whether a given object is an instance of a particular type. However, description logics do not provide a mechanism for constructing or performing calculations with objects.

The SYM calculus extends description logics into a full-featured programming language which is capable of dealing with both types and objects. The ideas in SYM first appeared as part of the Ohmu programming language,[9] but have now been refined into a formal object calculus.

There is one important operator which description logics lack, and that is a means to directly read the value of a slot. For instance, if we know that Mary is an instance of Mother, then we might like to find Mary's child, which I will write as Mary • child. In a normal DL ontology, these sorts of operations can be emulated by using the "∘" operator in conjunction with the "⊆" operator. For example, "parents who only spoil their grandchildren" can be written as Parent ⊓ spoils ⊆ child ∘ child. This mechanism is reasonably flexible, but it is not convenient for our purposes, primarily because it makes it hard to define normal forms. (See below for more details). Instead of expressing such a constraint as an equation involving "∘" and "⊆", we need to express it as a slot that uses "•" — e.g. Parent ⊓ $\mu X.\forall$spoils.X • child • child.

The "•" operator is a bit unusual because roles are not functional relations. Mary may have many children, so Mary • child refers to a set of objects rather than a single object. Since concepts denote sets, though, we can wrap that set up as another concept without running into any problems. The same principle can be applied to an abstract concept like Mother; Mother • child returns Human.

In other words, instead of treating roles as non-functional relations between objects, we can treat them as functional relations from concepts to concepts. This model does not distinguish between types and objects; an object is just a singleton type (i.e. a type with only one element). Instead of stating that a given object is an *instance* of a particlar type, subsumption is used to state that the object is a *subtype* of that type. I will refer to types in SYM as "prototypes" from now on, to distinguish them from traditional notions of types.

3.1. Grammar

The basic grammar for SYM is defined below. I have tried to use notation that more closely resembles mainstream OO languages, rather than the somewhat arcane syntax of DLs.

Syntax	Description	DL equivalent
$x, y, z ::=$	Self label	
$l, m, n ::=$	Slot label	Roles
$t, u, v ::=$	Term	Concept
$\quad \top, \bot, \mathsf{A}, x$	top, bottom, atomic types, and self labels	$\top, \bot, \mathsf{A}, X$
$\quad \{x \mid \overline{d}\}$	structure definition	concept in normal form
$\quad t.l$	selection	$C \bullet R$
$\quad t \mathbin{\&} u$	type intersection	$C \sqcap D$
$c, d, e ::=$	Slot	
$\quad l \; : \; t \; ;$	upper bound	$\forall R.C \;\sqcap\; \exists R$
$\quad l \; = \; t \; ;$	final binding	
$\quad l : \; t \Rightarrow u \; ;$	field	

A programming language reduces terms to values, where a value is a term in normal form – a term which cannot be reduced any further.[10] In a description logic, such terms take the form $C_1... \sqcap ...C_n$, where each C_i is either A, $\forall R.D$, $\exists R$, or $\mu X.\forall R.D$. Duplicate Rs are not allowed, so $\forall R.C \;\sqcap\; \forall R.D$ reduces to $\forall R.(C \;\sqcap\; D)$.

In the SYM calculus, the term $\{x \mid d_1..d_n\}$ defines a *structure*, which is a concept in normal form. (The notation $\overline{d}$ is shorthand for $d_1..d_n$). A structure consists of a number of slots, and each slot is both existentially and universally quantified. Combining the $\forall R.C$ and $\exists R$ constructs is a notational convenience that is appropriate for a programming language. The reason is that $\bot$ stands for the empty type – a type with no valid instances. An expression which reduces to $\bot$ thus indicates an error which should be caught by the static type system.

All slots in a structure share the same self-label x, which acts like the **this** keyword in C++ or Java. The fixpoint semantics in SYM differs somewhat from traditional DLs; in a description logic the label X refers to self-type of the class; in SYM the label x is the self-prototype, which can be either an abstract type or a singleton object.

The last two slot forms are extensions that flesh out the calculus as a programming language. A slot declaration of the form $l = t$ is called a *final binding*, and it has the following meaning in terms of sets: $\{a \in \Delta^I \mid \forall b.\ b \in t^I \text{ iff } (a, b) \in l^I\}$. In other words, a final binding not only places an upper bound on a role, it completely defines the role. This capability is required in order to properly define concrete objects.

A slot of the form $l : t \Rightarrow u$ declares a field/instance variable, which has essentially the same meaning as in Java. The term t is the type or range of the variable, while u is the value, which is constrained to be a subtype of the range. The range of a field is invariant, but the value can be overridden.

Fields are important because they allow the interface of a class to be separated from its implementation. When two structures are examined for type equivalence, only the ranges of fields matter; the values are ignored. This means that field values are effectively invisible when performing static type checking and logical reasoning; they only come into play when code is actually executed at run time.

Reduction: Atomic concepts and structures are already in normal form, so they can't be reduced further. Selection is implemented through standard fixpoint-style beta-reduction, where the variable x is bound to the structure declaration that it appears in. A formal definition is given by the E-PROJBETA rules in the appendix.

The only other term that needs to be reduced is type intersection. The intersection of two structures v_1 and v_2 contains all the slots in v_1, and all the slots in v_2. (Each slot represents a constraint on the type, so the intersection of two types can be calculated by taking the union of all their constraints.) Duplicate slot names are not allowed, so if both v_1 and v_2 have a slot with the same label, then the two slots are recursively combined: e.g. $\{l : t;\} \mathbin{\&} \{l : u;\} \longrightarrow \{l : t \mathbin{\&} u;\}$.

The intersection of an upper bound $l : t$, and a final bound $l = u$, is valid only if u is a subtype of t. The intersection of two final bounds $l = t$ and $l = u$ is only valid if $t = u$, for obvious reasons.

When two fields are combined, the value on the right overrides the value on the left: e.g. $\{l : u \Rightarrow t_1;\} \mathbin{\&} \{l : u \Rightarrow t_2\} \longrightarrow \{l : u \Rightarrow t_2\}$. The ranges of the two fields must be equivalent. This mechanism mimics standard overriding semantics, and thus allows implementation inheritance. Overriding does have an important consequence, though, which is that concept intersection is no longer entirely commutative; $t \mathbin{\&} u$ is not necessarily

equal to u & t. However, concept intersection is still commutative with respect to type equivalence (i.e t & u $\equiv$ u & t), because the ranges of fields are ignored when comparing structures for equivalency.

Functions: Much like other object calculi,[11,12] functions can be emulated as structures that contain an argument and a result. The function $\lambda arg : t.u$ can be written as $\{x \mid arg : t; result = u;\}$ Calling the function involves two steps — first create an activation record by using type intersection to bind the argument, and then select the result. The function application (f v) can be written (f & {arg = v;}).result. If v is a subtype of t, then the intersection (f & {arg = v;}) will reduce to the value {x | arg = v; result = t;}. If v is not a subtype, then the intersection will reduce to $\perp$, thus triggering a type error.

Here is a more concrete example. For clarity, the following assumes an extended calculus that defines atomic concepts for Int and the integer literals, and provides appropriate definitions of the arithmetic operators.

```
foo = { f | arg : Int ;   result = 2*f.arg ; };   // λarg : Int . 2*arg
bar = ( foo & { arg = 5 ; } ). result ;            // bar = ( foo 5 )
```

This syntax for declaring and calling functions is obviously a bit unwieldy, but SYM is a formal calculus, not a real programming language. Its objective is to provide as much power as possible within a minimal framework. A real language would obviously provide syntactic sugar for declaring and calling functions in a more reasonable way; please refer to the Ohmu programming language for a concrete example.[9]

Virtual Methods: Virtual methods are defined much like ordinary functions, except that both the argument and the result must be declared as fields.

```
MyClass = {
    foo : { f | arg : Int ⇒ 0; result : Int ⇒ 2*f.arg ; };   // λarg : Int . 2*arg
    bar = ( foo & { arg : Int ⇒ 5 ; } ). result ;            // bar = ( foo 5 )
}
```

Because foo.result is declared as a field, it can be overridden by derived classes.

The argument arg must also be a field for reasons related to static type safety. If the argument was declared using an upper bound then the function would be covariant, which is known to be unsafe.[7,13] But since the range of a field is invariant, the type of the foo.arg is also invariant, and type safety is preserved. Note that although the range of a field is invariant, the range may refer to *virtual types* elsewhere in the class, so it is still possible to express covariant concepts. I give an example a little bit later.

Classes and Inheritance: Classes are declared as one might expect. Here's how a simple Point class would be declared and instantiated:

```
Point   =           { x : Int ; y : Int ; };
origin = Point & { x = 0;  y = 0;  };
```

Both inheritance and instantiation are implemented via type intersection, so inheritance looks exactly the same:

```
Color        = { red : Int ; green : Int ; blue : Int ; };
ColorPoint = Point & { color : Color ; };
```

The prototype model also makes it possible to partially instantiate classes:

```
XPoint = Point & { y = 0; };   // points on the X axis
YPoint = Point & { x = 0; };   // points on the Y axis
origin = XPoint & YPoint;      // intersection of the two axes
```

Here XPoint and YPoint specialize Point by binding only one of the two variables. As one might hope, the origin can be defined as the intersection of the points on the x axis, and the points on the y axis. The origin is a single point (a singleton type) because all of its slots are final bound.

Generics Since classes and methods are both implemented with structures, it's easy to see that if SYM supports virtual methods, then it must also support virtual classes, which are also called virtual types. Virtual types, in turn, are one way to implement generics.[14,15,16,1] For example, here's a simple polymorphic list class:

```
List      =           { lst | item : T;     ThisClass : List ;   next : lst . ThisClass ⇒ ⊥ ; };
IntList = List & { lst | item : Int ;   ThisClass : IntList ; };
```

This example shows how a parameterized list class can be declared in a somewhat different way than in mainstream OO languages. The type of item itself does not need to be parameterized; it can simply be specialized using type intersection. However, we want to make sure that all nodes in the list have an equivalent type, and we do that by making the range of next equal to lst.ThisClass, where ThisClass is a virtual type that is specialized appropriately.

4. DISCUSSION

The model of inheritance presented here is substantially different from that provided by mainstream OO languages. Inheritance in SYM is implemented entirely by means of type intersection, which is inherently a multiple inheritance model. Classes in SYM are thus like traits or mixins.[4,17] Moreover, the intersection operator is recursive; it will will structurally weave two classes together by recursively descending into the tree of nested sub-structures. This mechanism can handle both virtual methods and Beta-style virtual classes.

Interestingly enough, this recursive weaving is essentially identical to Dr. Batory's feature composition operator, which has been used quite successfully to automatically build and reconfigure extremely large programs in Java.[5] It is also remarkably similar to the notion of aspect-weaving in aspect-oriented programming, although it lacks the quantifiers provided by AOP.[3] Both features and aspects are mechanisms for encapsulating *cross-cutting-concerns*, which are parts of the design that become tangled throughout multiple classes. The recursive nature of type intersection makes it to easy to encapsulate changes to multiple classes, which means that cross-cutting concerns are not nearly so big a problem.

It has been argued in the literature that there is a conflict between *interface inheritance*, which is also known as "subtyping", and *implementation inheritance*, which is sometimes called "subclassing".[8] The SYM calculus demonstrates that this conflict can be resolved quite simply. The interface of a class is defined using description logics, and inheritance preserves interfaces. The implementation of a class is defined by field values, which are hidden from the underlying logic, but which can still be overridden by the intersection operator. Interface and implementation inheritance need not and should not be separated.

Method overriding can create conflicts with multiple inheritance when two base classes try to override the same method, i.e. the infamous "diamond problem". Such overriding is predictable in SYM because the type intersection operator preserves the order in which classes are composed. A series of intersections thus creates a stack of mixin layers,[18] which are another way to implement features.

SYM is also unique because the language of types is identical to the language of objects. Both classes and objects are modeled as concepts, so any operation on one can be applied to the other. The compile-time type system has exactly the same capabilities as the run-time interpreter. This model stands in sharp contrast to traditional approaches, which provide a much more limited set of operations on types.[10]

4.1. Open Questions

The most important theoretical question about this model is decidability. Type safety in a description logic boils down to the question of *satisfiability*; we wish to know whether a particular concept composition is non-empty. Satisfiability is known to be undecidable for $\mu\mathcal{FL}^-(\subseteq, \circ)$. This is hardly unexpected. Since the type system and the run-time language are the same, decidability would mean that the language was not Turing complete.

There are three ways out of this dilemma. The first is to reduce the expressiveness of the type system, for instance by requiring that recursive calls be placed only in field values, where the type checker cannot see them. It is unclear at this point whether decidability can be achieved in this way without breaking the calculus.

The second option is to use incomplete reasoning. With incomplete reasoning, a "yes" answer to the satisfiability question means that a construct is type-safe, but a "no" does not necessarily mean that it isn't. This strategy has been used successfully in other expressive knowledge representations systems, such as LOOM.[19]

The third option is to supply a debugger that steps through the compilation process just as it steps through a running program. This is a viable option because the run-time and compile-time languages are the same.

REFERENCES

1. E. Ernst, "Family polymorphism," *Proceedings of ECOOP*, 2001.
2. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler, "Making the future safe for the past, adding genericity to the java programming language," *Proceedings of OOPSLA*, 1998.
3. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," *Proceedings of ECOOP*.
4. M. Flatt, S. Krishnamurthi, and M. Felleisen, "Classes and mixins," *ACM Symposium on Principles of Programming Languages*, 1998.
5. D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *Proceedings of ICSE*, 2003.
6. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. P.-S. editors, *The Description Logic Handbook, Theory, Implementation, and Applications*, Cambridge University Press, 2003.
7. K. B. Bruce, M. Odersky, and P. Wadler, "A statically safe alternative to virtual types," *Proceedings of ECOOP*, 1998.
8. K. Bruce, A. Fiech, and L. Petersen, "Subtyping is not a good "match" for object-oriented languages," *Proceedings of ECOOP*, 1997.
9. D. Hutchins, "The power of symmetry: Unifying inheritance and generative programming," *OOPSLA Companion, DDD Track*, 2003.
10. B. Pierce, *Types and Programming Languages*, MIT Press, 2002.
11. M. Abadi and L. Cardelli, *A Theory of Objects*, Springer, 1992.
12. M. Odersky, V. Cremet, C. Roeckl, and M. Zenger, "A nominal theory of objects with dependent types," *Proceedings of ECOOP*, 2003.
13. M. Torgersen, "Virtual types are statically safe.," *5th Workshop on Foundations of Object-Oriented Languages*, 1998.
14. O. Madsen and B. Møller-Pedersen, "Virtual classes: A powerful mechanism in object-oriented programming," *Proceedings of OOPSLA*, 1989.
15. O. Madsen, B. Møller-Pedersen, and K. Nygaard, *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993.
16. K. K. Thorup and M. Torgersen, "Unifying genericity — combining the benefits of virtual types and parameterized classes," *Proceedings of ECOOP*, 1999.
17. N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black, "Traits: Composable units of behavior," *Proceedings of ECOOP*, 2002.
18. Y. Smaragdakis and D. Batory, "Implementing layered designs with mixin layers," *Proceedings of ECOOP*, 1998.
19. "Loom website," *http://www.isi.edu/isd/LOOM/LOOM-HOME.html*.

5. APPENDIX – OPERATIONAL SEMANTICS FOR SYM

The small-step evaluation semantics for SYM are given below. There are two kinds of reduction. A type reduction, written $\xrightarrow{\mathcal{T}}$, is used at compile-time to calculate the types of terms. Value reduction, written $\xrightarrow{\mathcal{V}}$, is used at run-time to perform the actual computations. Value and type reduction are identical for all rules except E-PROJBETA, which differ in their handling of fields. The variable α is used to range over both $\mathcal{T}$ and $\mathcal{V}$.

Notation:

label(d) refers to the slot-label l associated with the slot d.

dom($\overline{d}$) refers to the set of all labels for $\overline{d}$.

$\overline{d} \ni e$ means that e is one of the slots in $\overline{d}$.

$\overline{d} \; \overline{\&} \; \overline{e}$ means $\overline{d} \& e_1 ... \& ...e_n$.

The judgement $t \prec: u$ refers to subtyping axioms that are true by definition, such as $t \prec: \top$, or $5 \prec: \mathsf{Int}$.

$$\frac{t \xrightarrow{\alpha} t'}{t.l \xrightarrow{\alpha} t'.l} \tag{E-PROJ}$$

$$\frac{\overline{d} \ni \text{ one of } \begin{cases} l:t; \\ l=t; \\ l:u \Rightarrow t; \end{cases}}{\{x \mid \overline{d}\}.l \xrightarrow{\mathcal{V}} [x \mapsto \{x \mid \overline{d}\}]\, t} \qquad \frac{\overline{d} \ni \text{ one of } \begin{cases} l:t; \\ l=t; \\ l:t \Rightarrow u; \end{cases}}{\{x \mid \overline{d}\}.l \xrightarrow{\mathcal{T}} [x \mapsto \{x \mid \overline{d}\}]\, t} \tag{E-PROJBETAV/T}$$

$$\frac{t \prec: u}{t \& u \xrightarrow{\alpha} t} \qquad \frac{u \prec: t}{t \& u \xrightarrow{\alpha} u} \tag{E-ISECTAXIOM1/2}$$

$$\frac{t \xrightarrow{\alpha} t'}{t \& u \xrightarrow{\alpha} t' \& u} \qquad \frac{u \xrightarrow{\alpha} u'}{v \& u \xrightarrow{\alpha} v \& u'} \tag{E-ISECT1/2}$$

$$\frac{\overline{e} = [x \mapsto z]\overline{c} \; \overline{\&} \; [y \mapsto z]\overline{d} \quad z \text{ chosen fresh}}{\{x \mid \overline{c}\} \& \{y \mid \overline{d}\} \xrightarrow{\alpha} \{z \mid \overline{e}\}} \tag{E-ISECTREC}$$

$$\frac{\text{label}(e) \notin \text{dom}(\overline{d})}{\overline{d} \& e \xrightarrow{\alpha} \overline{d}, e} \qquad \frac{\text{label}(d_i) = \text{label}(e)}{\substack{d_{1..i-1}, d_i, d_{i+1..n} \& e \xrightarrow{\alpha} \\ d_{1..i-i}, (d_i \& e), d_{i+1..n}}} \tag{E-MERGE1/2}$$

$$
\begin{array}{llll}
l:t; & \& \quad l:u; & \xrightarrow{\alpha} \quad l: t \& u; & \text{(E-ISLOT1)} \\
l=t; & \& \quad l=t; & \xrightarrow{\alpha} \quad l=t; & \text{(E-ISLOT2)} \\
l:t \Rightarrow u; & \& \quad l:t' \Rightarrow u'; & \xrightarrow{\alpha} \quad l:t \Rightarrow u'; \quad \text{if } t' \equiv t & \text{(E-ISLOT3)} \\
\\
l:t; & \& \quad l=u; & \xrightarrow{\alpha} \quad l=u; \qquad \text{if } u <: t & \text{(E-ISLOT4)} \\
l=u; & \& \quad l:t; & \xrightarrow{\alpha} \quad l=u; \qquad \text{if } u <: t & \text{(E-ISLOT5)} \\
l:t; & \& \quad l:u \Rightarrow s; & \xrightarrow{\alpha} \quad l:u \Rightarrow s; \quad \text{if } u <: t & \text{(E-ISLOT6)} \\
l:u \Rightarrow s; & \& \quad l:t; & \xrightarrow{\alpha} \quad l:u \Rightarrow s; \quad \text{if } u <: t & \text{(E-ISLOT7)}
\end{array}
$$

The operational semantics given here describe both the reduction of terms, and the typing of terms. Type safety is easy to prove because the type judgement $t \xrightarrow{\mathcal{T}} u$ and the reduction rules $t \xrightarrow{\mathcal{V}} u'$ differ in only one rule. The semantics for judging satisfiability and subsumption require a bit more explanation, and so are not included here.

38

An anomaly of subtype relations at component refinement, and a generative solution in C++

Zoltán Porkoláb[a] and István Zólyomi[a]

[a]Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary

ABSTRACT

Separation of concerns and collaboration based design is a good design choice: it results an easily maintainable and readable code. After separating orthogonal functionalities we assemble the required concerns as needed. However in real life, components could be used only after appropriate refinement steps, thus orthogonal concerns form independent specialization hierarchies. Such hierarchies provide individual subtype relations. The specific solution for a particular task can be finally produced by composing a set of classes from these refinements. However, a subtype anomaly occurs between collaborating groups having different number of participating classes from different refinement stages. In this article we walk around this anomaly we called chevron-shape inheritance and present a framework to handle collaborating groups of classes using template metaprogramming based on standard C++ features.

Keywords: Multiple inheritance, structural subtyping

1. INTRODUCTION

The creation of large scale software systems is still a critical challange of software engineering. Several design principles exist to keep the complexity of large systems manageable. Different methodologies are used to divide the problem into smaller orthogonal parts that can be planned, implemented and tested separately with moderate complexity. In a fortunate case such parts are already exist in some foundation library, otherwise they can be produced by reasonable efforts. This *separation of concerns* is widely discussed in[17] and.[20] In object-oriented systems these concerns are mostly implemented in separate classes.

Having premanufactured components we have several methodologies to assemble a full system from the required code parts. This so-called *collaboration based design* is supported by aspect oriented programming,[15] composition filters,[11] subject oriented programming[1921] and HyperJ.[18] Besides, the assembly can be naturally expressed using multiple inheritance by deriving from all required components in languages supporting this feature such as C++. This *mixin-based* technique is highly attractive for implementing collaboration-based design.[6] Whichever approach we choose, the basic idea is to create a union of the interfaces of the collaborating classes. These classes represent orthogonal concepts thus the result is a disjunction of their functionalities.

However, in real life it is hard to find a component that represents the required concept *exactly*. In most cases we have to customize the components to fit the needs of the current task. Specializations for every separate concern are made independently which leads to separated specialization hierarchies, each representing refinement steps of an individual concept. The specific solution for a particular task can be finally produced by assembling specialized concepts from appropriate levels of different hierarchies. In object-oriented languages we mostly represent our concepts as classes. Specializations are regurarly expressed using inheritance, hence we gain a subtype relationship between the refined and the original component.

The problem appears when we try to refer to a subset of supertypes of the collaborating classes. It is a desired feature because a client code should be separated from the knowledge of the exact type of the (refined) components. But the collaboration of original components are not a base class of the collaboration of refined components. Because automatic conversion is out of order, objects of the collaboration of original components

---

Further author information: (Send correspondence to Zoltán Porkoláb)
Zoltán Porkoláb: gsd@elte.hu, István Zólyomi: scamel@elte.hu

cannot be used in place of objects of collaboration of refined components which is against the Liskov Substitution Principle.[22] Similarly, clients are unable to utilize dynamic binding calling functions of derived objects in a type safe way.

2. IMPORTANCE OF THE PROBLEM

Programmers may argue that such side effects may ever appear in practice. In this section we intend to convince the reader showing real-life examples.

We start with one from the C++ Standard Library of C++: in figure 1 you can see the stream class hierarchy of the standard library[*].
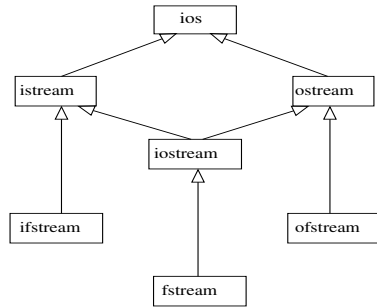
Figure 1. I/O library according to the C++ standard

Classes `istream` and `ostream` are representing input and output streams as orthogonal concerns. (There is a common base class `ios` for both classes holding some general stream functionality.) Class `iostream` is created using multiple inheritance unifies input and output functionalities representing streams that can be both read and written[†]. The library contains two refinements of both input and output stream concepts. Streams opened over certain physical devices belong to classes `ifstream` or `ofstream` as refinements of `istream` and `ostream` respectively[‡]. These specializations are implemented using inheritance. Class `fstream` (and `stringstream` also) inherits from `iostream` and represents file streams for both input and output operations.

Surprisingly, this construction causes some unexpected results. `fstream` is clearly a subtype of both `ifstream` and `ofstream`. The inheritance hieararchy described above does not express this, hence there is no conversion from `fstream` to neither `istream` nor `ostream`. Clients handling input files are not able to use objects from `fstream` as an instance of `ifstream`, they are enforced to use `istream` as a more general interface losing file specific information. After taking a look at classes `iostream` and `istream` this fact may be an astonishing fact.

The other example is from the programming language Eiffel.[10] The kernel library of Eiffel contains several abstract classes like `NUMERIC` for arithmetics, `COMPARABLE` for sorting, `HASHABLE` for associative containers, etc. These classes are practical to have because in Eiffel we can require a template parameter to be a subclass of such an "interface". These classes can be combined as needed using multiple inheritance, hence we can derive a `NUMERIC_COMPARABLE_HASHABLE` or a `NUMERIC_COMPARABLE` interface directly from the bases. Again, the problem appears when we try to use an object of the first class with a generic algorithm requiring the latter type: no subtype relation is realized, we have to resolve it by hand creating funtions for conversion.

3. THE CHEVRON-SHAPE ANOMALY

In this section we formulate the problem showing a general description and suggest a name for the anomaly. It appears in strongly typed object-oriented languages which base their subtype relation on inheritance; conse-

---

[*]We omit the fact that all the followng classes are *templates* by the standard, because this does not affect our problem.

[†]This results in a known anomaly called *diamond-shape inheritance*. In this case it is resolved using *virtual inheritance* in classes `istream` and `ostream`.

[‡]Similar specializations exist for streams stored in a memory buffer (e.g. `istringstream` and `ostringstream`).

quently it appears in all widely used object-oriented languages, such as Java, C++, C#, Eiffel, Object Pascal, etc. The problem is closely related to class refinement using multiple inheritance[§].

Assume we have a set of independent base classes implementing orthogonal concerns. These classes are to be refined stepwise, thus each concept forms a separate inheritance hierarchy. The solution for a specific user requirement can be constructed as a group of refined concerns. In the same case, we should be able to use any subset of classes from these hierarchies as interfaces to the previously constructed group. Therefore subtype relation should stand between any of these groups irrespectively of the number and refinement level of participant concern classes. The subtype relation should be closed under union (disjunction), but this is not fulfilled in object-oriented languages. Thus we have to decide: if we derive the refined collaboration from the original collaboration class we lose the subtype relationship with the refined bases; otherwise (deriving from the refined bases) we lose the subtype relationship with the original collaboration. In most design cases the latter situation is preferred. In figure 2 the general structure of the anomaly can be seen according to the two mentioned cases respectively. In the picture missing subtype relations are marked with dashed lines.
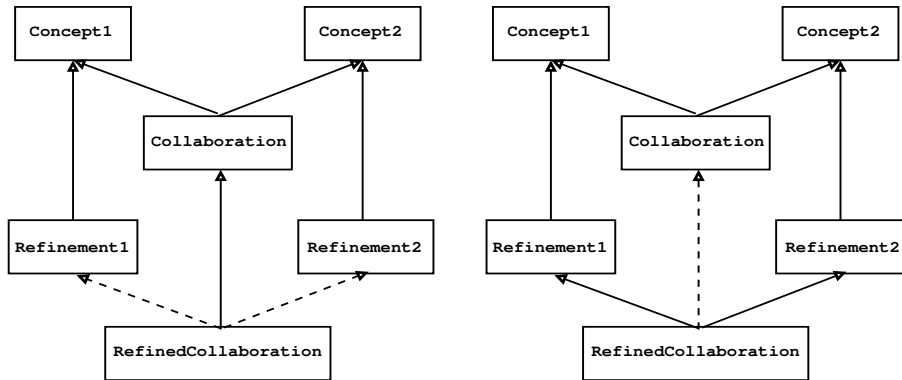

Figure 2. Chevron-shape inheritance

We gave the name *chevron-shape inheritance* to this anomaly[¶]. It is easy to understand our choice taking a look at figure 2.

In addition, having several concept hierarchies we should be able to express subtype relationship between collaborating groups having different number of classes of different refinement stages.

4. CSET

Besides its object-oriented tools the C++ language also has a rich feature set for supporting generative programming with templates. Theoretically template metaprogramming in C++ is a Turing complete language itself, therefore any algorithm can be expressed as a metaprogram[‖] (see[7]). This "language" is "executed" in compilation time: the result is a C++ program which is still about to be checked by the language strong type system.

Template metaprogramming features discussed above make us able to solve the chevron-shape anomaly. To achieve this goal we have to simulate a subtype relationship between adequate sets of collaborating classes: based on the possibilities of template metaprogramming we implement conversions between the sets[**]. Presenting the technical implementation details is out of the scope of this paper. The main issue in `CSet` is to build the needed class hierarchy, templated conversion operators and smart pointers automatically in compilation time.

[§]Note that some languages (e.g. Java) do not support multiple inheritance directly, but are able to simulate it (e.g. using interfaces). The problem exists in these cases, too.

[¶]With this name we also intended to refer to *diamond-shape inheritance.*

[‖]Practically compilers have limitations in resources (e.g. a maximal depth of recursion during template instantiation).

[**]We call these sets `CSets` where `C` can be pronounced as any of class, concern, collaboration, chevron, etc as conceptually needed.

5. SUMMARY AND RELATED WORKS

The subtyping mechanism of current object oriented languages is not flexible enough to express required subtype relationships arising at implementation of collaboration based designs. We described an anomaly called chevron-shape inheritance which arises assembling sets of collaborating concerns created in stepwise refinement of concepts. We introduced a framework called CSet based on C++ template metaprogramming to extend the possibilities of subtyping mechanism between sets of collaborations. CSets make the subtype relation disjunctive with respect to multiple inheritance. It supports coercion polymorphism between appropriate collaborating groups or inclusion polymorphism allowing dynamic binding of methods with smart pointers. The framework is strictly based on standard C++ features, therefore neither language extensions nor additional tools are required.

Another candidate for solution can be the signature facility of C++ from Gerald Baumgartner.[4] Signatures provide a facility similar to interfaces in Java, but in a non-intrusive way: if a class intends to implement a signature, it does not have to define it explicitly to do so. Signatures are non-standard language extensions and are implemented only in g++ compilers, thus their usability is strictly limited.

As an alternative solution Structural subtyping[13] provides an excellent possibility for solution: languages supporting this feature do not suffer from our anomaly. Unfortunately no widely used object-oriented language provides structural subtyping.

REFERENCES

1. István Zólyomi, Zoltán Porkoláb, Tamás Kozsik: An extension to the subtype relationship in C++. GPCE 2003, LNCS 2830, pp. 209 - 227, 2003 (Springer-Verlag Berlin Heidelberg)
2. Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
3. David Vandevoorde, Nicolai M. Josuttis: C++ Templates: The Complete Guide. Addison-Wesley (2003)
4. Gerald Baumgartner, Vincent F. Russo: Implementing Signatures for C++ ACM Transactions on Programming Languages and Systems (TOPLAS) Vol. 19 Issue 1. 1997. pp. 153-187.
5. Todd Veldhuizen: Using C++ Template Metaprograms. C++ Report vol. 7, no. 4, May 1995, pp. 36-43.
6. Yannis Smaragdakis, Don Batory: Mixin-Based Programming in C++. In proceedings of Net.Object Days 2000 pp. 464-478
7. Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)
8. Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)
9. Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley (1994)
10. Bertrand Meyer: Eiffel: The Language. Prentice Hall (1991)
11. Lodewijk Bergmans, Mehmet Aksit: Composing Crosscutting Concerns Using Composition Filters. Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.
12. Kim B. Bruce: Foundations of Object-Oriented Languages. The MIT Press, Cambridge, Massachusetts (2002)
13. Luca Cardelli: Structural Subtyping and the Notion of Power Type. Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, January 1988. pp. 70-79.
14. Erik Ernst: Family Polymorphism. in Proceedings ECOOP 2001, Budapest, Hungary, Springer-Verlag LNCS 2072, 2001 pp. 303–326,
15. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin: Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241, June 1997.
16. Ulrich W. Eisenecker, Frank Blinn and Krzysztof Czarnecki: A Solution to the Constructor-Problem of Mixin-Based Programming in C++. Presented at the GCSE2000 Workshop on C++ Template Programming.

17. Harold Ossher, Peri Tarr: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. IBM Research Report 21452, April, 1999. IBM T.J. Watson Research Center. http://www.research.ibm.com/hyperspace/Papers/tr21452.ps

18. Harold Ossher, Peri Tarr: Hiper/J. Multidemensional Separation of Concerns for Java. International Conference on Software Engineering. ACM pp. 734-737. 2001

19. William Harrison, Harold Ossher: Subject-oriented programming: a critique of pure objects Proceedings of 8th OOPSLA 1993, Washington D.C., USA. pp. 411-428. 1993

20. Don Bathory, Jia Liu, Jacob Neal Sarvela: Refinements and multi-dimensional separation of concerns Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering. helsinki, Finland, 2003.

21. Subject Oriented Programming. http://www.research.ibm.com/sop

22. Barbara Liskov: Data Abstraction and Hierarchy SIGPLAN Notices. 23(5), May 1988

23. Jonathan E. Shopiro: An Example of Multiple Inheritance in C++: a Model of the Iostream Library. ACM SIGPLAN Notices, December, 1989

Java with Traits —
Improving Opportunities for Reuse

Philip J. Quitslund and Andrew P. Black

OGI School of Science & Engineering
Oregon Health and Science University
Portland, Oregon USA

ABSTRACT

The Java language includes features that present significant barriers to reuse; in practice, programmers have no choice but to copy and paste code that is not accessible via inheritance. Traits improve code-sharing in Smalltalk by providing a means to reuse such behavior, and we claim that a similar mechanism for Java would overcome not just the lack of multiple inheritance but Java's other barriers to reuse as well, including the use of private, final and synchronized qualifiers. In support of our claim we present the initial findings of a case study of Java Swing, a large production-quality library, showing how we isolated pieces of duplicated code that could not be eliminated by conventional means and how traits could be used to eliminate them.

Keywords: Reuse, Traits, Java, Code Duplication, Java Swing

1. JAVA'S BARRIERS TO REUSE

Four features of Java are responsible for significant code duplication in the Java Foundation Classes (JFC).

1. **Lack of Multiple Inheritance of Implementation.** With Java interfaces we can group classes in different hierarchies by the protocols they support, effectively enabling *multiple inheritance of type.* This allows clients to treat objects with a shared protocol uniformly, irrespective of their representation. While this promotes flexibility and a generic style of programming, it does nothing to address reuse: in many cases, classes that share protocols would also like to share aspects of their implementations. As a concrete example, take the PrintStream and Printwriter classes from the java.io package. Both support a uniform printing protocol (print(boolean), print(int), print(long), and so on) and *identical* implementations for twelve methods. While the shared protocol can be reflected by an interface, Java has no feature to abstract out and share the implementation because PrintStream and Printwriter both subclass classes in which printing is not appropriate (FilterOutputStream and Writer, respectively).

2. **Inaccessible Private Inner Classes.** Inner classes are a useful mechanism for grouping related classes and for codifying "friend" relationships between classes. Their use is especially common in GUI applications where they provide a convenient means for implementing call-backs and adapters. Unfortunately, inner classes are also very difficult to reuse because the conventional wisdom is to make them private to negate the security risk that they introduce*. An example of this phenomenon can be seen in the java.util.concurrent package where the FutureTask and ScheduledExecutor classes define identical, but non-reusable, private inner ListIterator classes.

Send correspondence to: philipq@cse.ogi.edu (Philip Quitslund) or black@cse.ogi.edu (Andrew Black).

*The JVM does not support inner classes directly. Instead they are compiled into separate classes that gain access to their containing class's fields and methods via compiler-generated accessor methods. Effectively this promotes methods and fields to public that might otherwise be private. Although the accessors are "hidden" behind mangled names, a malicious programmer could craft bytecode that violates their intended encapsulation policies.

3. **Non-Extensible Final Classes.** Making a class final ensures that it cannot be subtyped (it might also improve performance). Because Java equates subtyping and subclassing, final restrictions also block opportunities for reuse. A canonical example is Java's representation of strings in java.lang. To ensure the proper functioning of the interpreter and compiler, which depend on its concrete implementation, the class String is declared final to prevent programmers from substituting subtypes that break its semantic contract. To reuse String's implementation one is forced to copy and paste (as in java.lang.AbstractStringBuilder, where parts of String's indexing behavior are duplicated).

4. **Synchronized Variations.** In some cases, a basic concurrent version of a class can be obtained by adding the synchronized modifier to the critical methods. Such is the case with Vector (synchronized) and ArrayList (unsynchronized) in java.util which could share, with a little refactoring, at least fourteen method bodies if we could selectively introduce synchronization.

2. WHAT ARE TRAITS?

Traits[1] are a mechanism for code reuse that complements single inheritance. Traits, like classes, are containers for methods. But, unlike classes, traits have no fields. Traits, like abstract classes, cannot be instantiated directly; instead, they are composed into classes (which are instantiable). A class ColorPoint might be composed of traits TColor and TPoint and other bits (like state, for example), which means that ColorPoint will have the methods defined in TColor and TPoint. Because method names might conflict, composition can be selective, allowing for the removal and renaming of composed methods. Thus, if TColor and TPoint both define an equal method, we can exclude either implementation or alias one or both with another name. If the conflict is left unresolved, the composition includes neither trait method but instead includes a special stub method indicating an unresolved conflict. To use this trait, the programmer must explicitly disambiguate the conflict by exclusion or by defining an overriding method in the client class.

Commonly, traits refer to methods that they do not themselves define. Traits that do not define all the methods they call are said to *require* these methods. A *comparable* trait, for example, might define comparison methods ($<=$, $>=$, $\sim=$, min, max, and so on) in terms of the $<$ and $=$ operations that it *requires* (see Figure 1). In order for a class to use this trait it must provide $<$ and $=$ methods. If it does not, it is considered incomplete (ostensibly abstract).
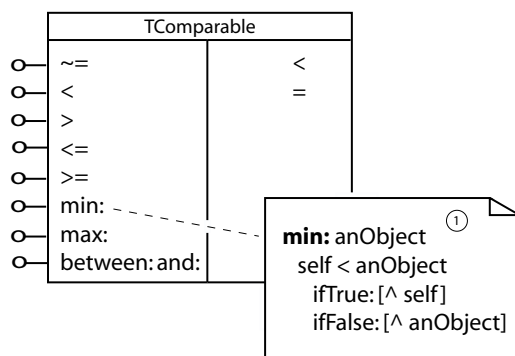

Figure 1. Trait TComparable *provides* $\sim=$, $<$, $>$, and so on (the methods on the left) and *requires* $<$ and $=$ (on the right). Provided methods can be implemented in terms of required ones as in the definition of min: (①).

2.1. Extending Traits for Java

In seeking to adapt the ideas behind traits to the Java language, we must consider several issues that do not arise with Smalltalk. Most obviously, Java methods and constructors frequently refer to their enclosing class by name, so it is necessary to provide a mechanism (e.g., a thisclass keyword) that trait methods can use to abstract away from any specific named class. Beyond this, the need to declare types for local variables and parameters

in methods may make methods from traits less reusable than they would be if they could be typed generically. In addition, we must consider features like nested classes, visibility modifiers (public, private, protected) and other modifiers such as final, synchronized, and native. Which of these features merit first-class treatment?

We believe that a simple extension of the trait composition clause, not unlike the the aliasing mechanism of Smalltalk traits, will be sufficient to resolve the code duplication problems found in the JFC. In Smalltalk traits, an alias expression such as TEnumerable @(map→collect) denotes a trait like TEnumerable except that it also contains an additional method callen map whose implementation is the same as that defined for collect. A similar mechanism can be used to adjust the modifiers on trait methods when they are incorporated into another trait or class. Unlike the alias mechanism, which simply adds a new name for an existing method, a modification mechanism would need to hide the old version of the method, and introduce a new one with the modified property (see Figure 2 for a possible syntax). If clients could add declaration modifiers when using methods defined in traits, the difficulties introduced by final classes and synchronization modifiers could be sidestepped. Such a mechanism would allow the same methods to be reused in final and non-final and synchronized and un-synchronized settings. Inner classes, though a bit more complex, could be made shareable in a similar way.

```
class MySynchronizedVector uses TVector@{∗ as synchronized;} { // ... }
```

Figure 2. A synchronized Vector variation declared using a pattern-matching scheme like that employed in AspectJ's pointcut language.[2] The @ operator indicates an alias operation and the wildcard matches all of TVector's methods.

3. A CASE STUDY: CODE DUPLICATION IN JAVA SWING

The Java Foundation Class (JFC) libraries are flush with examples of code duplication that cannot be eliminated by single inheritance. To provide more than anecdotal support for the value of traits we sought to quantify just how much duplication there can be in production systems. To evaluate how traits might improve code-sharing in the wild, we looked at Java Swing, a library of cross-platform GUI components provided with the Java distribution. We focused on duplication that seems to result from the restricted power of single inheritance. We chose Swing because it is production quality and quite large—in the JDK 1.5.0,[3] Swing consists of 605 top-level classes/interfaces and over 290 thousand lines of (commented) code. We obtained a conservative estimate of code duplication in Swing by using the freely available CPD ("Copy Paste Detector") tool,[4,5] which employs the fast (but naïve) Karp-Rabin string-matching algorithm.[6] CPD detected over 15 thousand duplicated lines across 231 classes, accounting for 5 percent of the source and 38 percent of Swing's classes.

Surprisingly, some of this duplication might be eliminated using standard features of Java, without the need for traits. That is, if classes C_1 and C_2 contain duplicated methods and also share a direct superclass, then the duplicated methods could possibly be raised to the shared superclass or put in an intermediate shared abstract superclass. Similarly, if code is multiply defined in a class and its superclass, then the copy in the subclass can be eliminated.

Candidates for traits are those cases where the classes sharing the behavior do not share an immediate superclass. Here the feasibility of removing the duplication with inheritance depends on how far the classes containing the duplication are below their lowest shared superclass—a metric we will call *inheritance depth*. We define the inheritance depth to be the sum of the distances between two compared classes and their shared superclass. Figure 3 shows three scenarios: if code is duplicated in a class and its superclass, then we say the depth is one (case a), if it is in classes that share an immediate superclass, then we say the depth is two (case b), and if one of two sibling classes is separated from the shared superclass by another class, then we say the depth is three (case c). The smaller the inheritance depth, the easier it is to remove the duplicated method. The duplicated paint method is trivial to remove in (a), straightforward in (b) but tricky in (c). The danger in putting paint in D_3 is that it may not be appropriate there or in C_5 which inherits it — here code is shared at the expense of understandability. This phenomenon has been described as putting behavior "too high"[7] and is a prime candidate for refactoring to use traits.
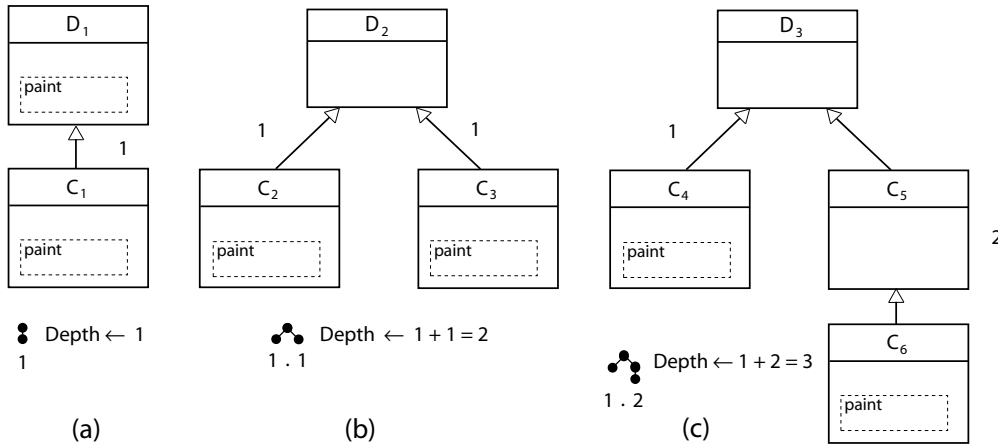
Figure 3. Duplicated methods and relative inheritance depths. Notice that depths can be represented as pairs that describe the shape of of the hierarchy. This helps differentiate between different hierarchies that share the same depth. For instance, the two distinct hierarchies that have depth 3 can represented by the pairs $\langle 3 \cdot 0 \rangle$ and $\langle 2 \cdot 1 \rangle$ (or its equivalent, $\langle 1 \cdot 2 \rangle$).

3.1. Results

To get a sense for how much of Swing's duplication is too deep to eliminate by single inheritance, we measured inheritance depths for 127 shared fragments accounting for over two thirds of the duplicated code. Of these cases we were surprised to find 58 where the code was duplicated within the same class or in an immediate superclass (case a) and 32 in sibling classes with a shared superclass (case b). Clearly duplication in Swing could be much reduced by traditional refactoring! The remaining 37 instances (or 29 percent) are prime candidates for traits. Figure 4 summarizes the distribution of inheritance depths for these candidates.
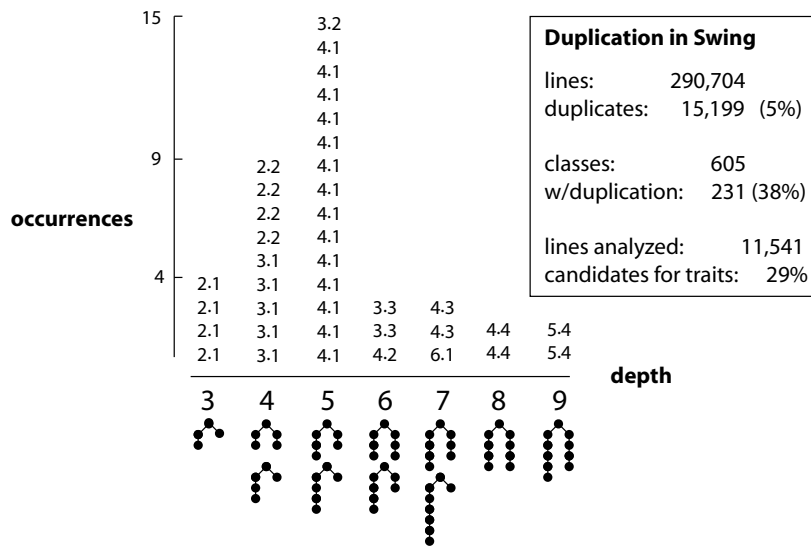

Figure 4. Distribution of inheritance depths in candidates for refactoring in Swing. For brevity, enclosing brackets are excluded from the depth pairs in the histogram. Thus, $2 \cdot 1$ in the first column should be interpreted as the pair $\langle 2 \cdot 1 \rangle$.

4. CONCLUSION

Smalltalk traits greatly reduce the need to copy and paste by providing a means to reuse behavior that is entirely separate from inheritance. In addition to lacking multiple inheritance, Java has other features that limit code-sharing. We believe that a well-designed mechanism for traits in Java could help us to overcome several of these obstacles. A naïve analysis of Swing detected 5 percent code duplication, of which at least 29 percent can be eliminated with traits but not by single inheritance. However, this is just the beginning. The CPD string-matching approach to finding duplication is extremely conservative and a more sophisticated algorithm would doubtless find more duplication. Moreover, code duplication says nothing of *logic duplication*. Our informal study of the JFC indicates that there is a great deal of logic duplication that is not detectable by such methods. Finally, it is worth noting that we only sought the most obvious opportunities to refactor to traits in looking for code that cannot be shared because single inheritance is insufficiently expressive. We believe that Java's other barriers to reuse (listed in Section 1) are responsible for a good deal more duplication, which traits could eliminate.

ACKNOWLEDGMENTS

REFERENCES

1. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable units of behavior," in *Proceedings of ECOOP 2003 - European Conference on Object-Oriented Programming*, *Lecture Notes in Computer Science* **2743**, Springer, 2003.

2. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings European Conference on Object-Oriented Programming*, *Lecture Notes in Computer Science* **2072**, pp. 327–353, Springer-Verlag, (Berlin, Heidelberg, and New York), 2001.

3. Sun Microsystems, "Download Java 2 Platform, Standard Edition 1.5.0 Beta 1." http://java.sun.com/j2se/1.5.0/download.jsp. (April, 2004).

4. T. Copeland, "Detecting duplicate code with PMD's CPD," *On Java*, Mar. 2003. http://www.onjava.com/pub/a/onjava/2003/03/12/pmd_cpd.html.

5. PMD, "PMD Project." http://pmd.sourceforge.net/. (April, 2004).

6. R. M. Karp and M. O. Rabin, "Efficient randomized pattern–matching algorithms," *IBM Journal of Research and Development* **32**, pp. 249–260, 1987.

7. A. P. Black, N. Schärli, and S. Ducasse, "Applying traits to the smalltalk collection classes," in *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 47–64, ACM Press, 2003.

Merging conceptual hierarchies using concept lattices

Mohamed H. Rouane[a] and Petko Valtchev[a] and Houari Sahraoui[a] and Marianne Huchard[b]

[a] DIRO, Université de Montréal, C.P. 6128, Succ. "Centre-Ville", Montréal, Canada, H3C 3J7;
[b] LIRMM, UMR 5506, 161 rue Ada, 34392 Montpellier cedex 5, France

ABSTRACT

In many situations, one faces the need for integrating a set of conceptual hierarchies that represent parts of the same domain. The target structure of the integration is a unique conceptual hierarchy that embeds at least the total of the knowledge encoded in the initial ones. The key issues to address here are the discovery of higher-level abstractions on top of the existing concepts and the resolution of naming conflicts among these. Formal concepts analysis (FCA) is a mathematical approach toward abstracting from attribute-based object descriptions which has been recently extended to fit relational descriptions thus giving rise to the relational concept analysis (RCA) framework. Building up on RCA, our integration approach amounts to encoding the initial hierarchies into set of binary tables, one for each component category, e.g., classes, associations, etc., and subsequently constructing, querying and reorganizing the corresponding abstraction hierarchies until a unique satisfactory hierarchy is obtained. This paper puts the emphasis on the mechanisms of discovering new abstractions and exporting them between the abstraction hierarchies of related component categories. The impact of naming conflicts within the RCA process is discussed as well. The paper uses UML as description language for conceptual hierarchies.

Keywords: Hierarchy integration, formal concept analysis, abstraction, naming conflict resolution.

1. MOTIVATION

Specialization hierarchies embody the knowledge about a particular domain expressed by means of concepts, concept features and inter-concept relationships (general kind relationships, composition, aggregation, etc.). The design of such hierarchies is known to be a hard-to-automate problem, even in the cases where the ground set of objects/classes is given and the goal is simply to organize them into a meaningful structure. Consequently, most methods that support the hierarchy design task apply some inductive techniques in a semi-automated or interactive mode.

We focus here on the integration of specialization hierarchies. In fact, the need for merging two or more existing specialization hierarchies into a unique global one that encodes at least the initial hierarchical knowledge occurs in many practical situations. Thus, in database design, it is known as *schema integration*,[1] in knowledge engineering as *ontology merge*,[2] in software modeling as *model integration*[3] (e.g., assembly of subjects or aspect inter-weaving), etc. Whatever the name given and the domain-specific constraints, the integration of hierarchies basically consists in assembling a set of potentially incomplete views on the same reality.

Yet the assembly is more than a mere juxtaposition of hierarchies as parts of these may overlap whereas overlapping between concepts of different hierarchies may remain partial, thus suggesting these are the variants of a more general concept. To sum up, integration requires the detection of overlapping parts of the initial hierarchies whereby some of the corresponding elements, i.e., concepts, relationships, properties, etc., stemming from different hierarchies will be directly recognized as identical whereas, others, although similar, will rather give rise to abstractions that have not been discovered beforehand.

In this paper, we discuss the automated support for the integration process which remains manual in its very nature. Indeed, identity recognition for elements of different hierarchies is ultimately up to the designer's judgment as concept naming is prone to errors and ambiguity, e.g., the same domain element may be given different, yet synonymous, names in different hierarchies. Moreover, the choice of the "interesting" abstractions among the set of all plausible ones is hardly automated. We sketch an approach for hierarchy integration that relies on abstraction mechanisms from the formal concept analysis (FCA)[4] field. FCA turns a binary (individuals

---

(rouanehm,valtchev,sahraouh)@iro.umontreal.ca, huchard@lirmm.fr

x properties) table into a complete lattice made up of pairs of closed sets. To cover more sophisticated concept descriptions, a version of FCA is used that feeds inter-individual relations into the generalization process. In our framework, mechanisms detecting regularities in individual/property co-occurrences and in the lattice structure are relied on to spot potential naming conflicts.

2. INTEGRATION CHALLENGES

In the remainder of the paper, we consider hierarchies described in UML.[5] Using the underlying terminology, *abstractions* represent generic domain concepts and are obtained by factorizing the shared specifications of classes or associations.

Figure 1 shows two class diagrams pertaining to strict subsets of the banking domain, i.e., the bank account sub-domain (henceforth identified by BA) and credit card sub-domain (CCA). Please notice that there is a substantial overlap between both in terms of business classes, associations and class members.
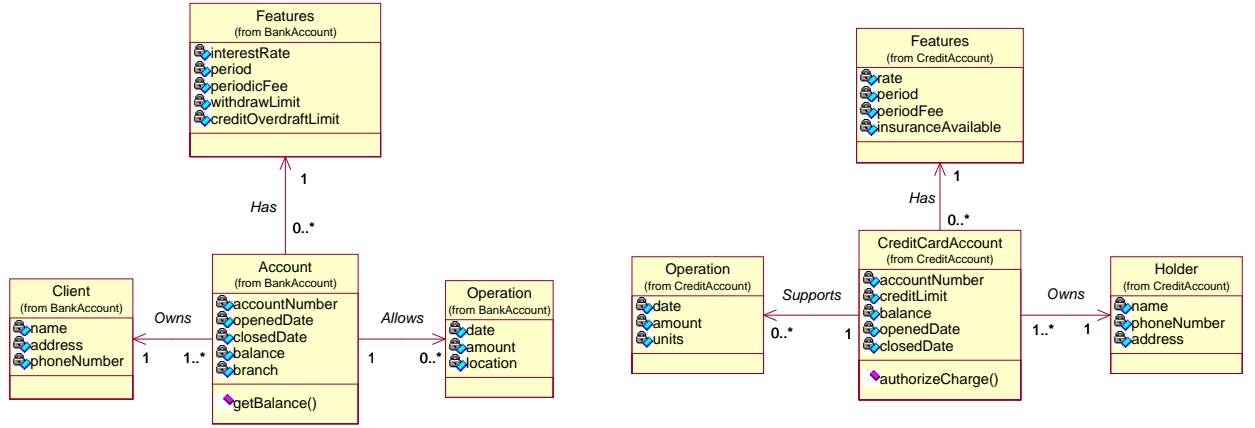

Figure 1. Left: Bank account sub-domain model. **Right:** Credit card sub-domain model.

When only a sub-domain is focused on, then obviously, the possibilities of identifying the appropriate generic concepts are limited. In particular, concepts transcending the sub-domain boundaries would be overlooked. For instance, although both hierarchies in Figure 1 are faithful representations of the underlying sub-domains, none of them identifies the general concept of account that would factor the specifications of `CreditCardAccount` and (check) `Account` (see class `Account` in Figure 6). These can only be discovered when the appropriate sub-domain hierarchies are put side-by-side.

Consequently, a clever integration approach should look for all the 'hidden' domain abstractions that cross the local hierarchy boundaries. As we shall show it in the following paragraphs, mathematical tools exist that, provided an appropriate encoding is performed beforehand, deliver the set of *all potentially useful abstractions* over a set of concrete entities.

There is yet another difficulty integration has to face which is due to the overlapping of some hierarchies. Indeed, restricted views and insufficient synchronization may cause mismatches in the naming of domain elements that happen to appear in several sub-domains, as the case of the `Client`/`Holder` classes suggests. Indeed, both `Client` and (card) `Holder` seem to reflect the same reality, i.e., bank customers whose only differentiation, except for naming, lays in the sort of banking products they possess. Hence, and despite the diverging class names, these classes should be merged in the global hierarchy (see Figure 6). In contrast, the pair of `Feature` classes, although identically named, rather translate variations of a hypothetic *Feature* entity. Again, the name (mis)match is due to the limited view in both local hierarchies: the lack of intra-hierarchy variation prevented the identification of the generic class `Feature` whose absence prompted its local variants (subclasses) to be named identically. It is noteworthy that in both above cases of misleading naming, the specification of the compared entities, i.e,, the list of attributes, is a strong indicator of whether these are identical or not: while both `Feature` classes share

52

only a subset of their properties, the description of `Client` and `Holder` match perfectly. In fact, integration is also a reconciliation activity on two inherently incomplete, i.e., partial, views on the same reality.

In the next section we present a domain that studies the construction of conceptual hierarchies from observation from a mathematical point of view and therefore provides a formal framework for the abstraction activity that is considered here.

3. FCA-BASED INTEGRATION

3.1. FCA basics

A context $\mathcal{K} = (O, A, I)$ is given where O is a finite set of formal objects, A is a finite set of formal attributes and $I \subseteq O \times A$ is a binary relation between these sets, saying that the object $o \in O$ has the attribute $a \in A$ whenever $(o, a) \in I$ holds.

	accountNumber	address	amount	balance	branch	closeDate	creditLimit	creditOverdraftLimit	date	insurance	interestRate	location	name	openedDate	period	periodicFee	phoneNumber	units	withdrawLimit
BA-Account	X			X		X								X					
Client		X											X				X		
BA-Features								X			X				X	X			X
BA-Operation			X						X			X							
CCA-Account	X			X		X	X							X					
CCA-Operation			X						X									X	
CCA-Features										X	X				X	X			
Holder		X											X				X		

Table 1. Binary context $\mathcal{K}^0_{class}$. Formal objects are UML classes and formal attributes are their variables.

Table 1 represents a context drawn from Figure 1. The image of an object set $X \in O$ is defined by: $X' = \{a \in A \mid \forall o \in O : (o, a) \in I\}$ and dually the image of an attribute set $Y \in A$ is defined by: $Y' = \{o \in O \mid \forall a \in A : (o, a) \in I\}$. For example within Table 1, $\{\text{BA-Account}, \text{CCA-Account}\}' = \{\text{balance}\}$ and $\{\text{withDrawLimit}\}' = \{\text{BA-Features}, \text{CCA-Features}\}$. A (formal) concept in $\mathcal{K}$ is a pair $(X, Y) \in \mathcal{P}(O) \times \mathcal{P}(A)$ for which $X = Y'$ and $Y = X'$ where X is called the *extent* and Y the *intent*. The set $\mathcal{C}$ of all concepts provided with an order relation based on extent inclusion, say $\mathcal{L}$, is a complete lattice. The lattice drawn from the context of Table 1 is shown in Figure 2 on the right. As many practical applications involve non-binary data, *many-valued contexts* have been introduced in FCA. In such a context $\mathcal{K} = (O, A, V, J)$, an object o is described by a set of attribute value pairs (a, v), meaning that J is a ternary relation that binds the objects from O, the attributes from A and the values from V (see table presented on the left-hand side of Figure 2). The construction of a lattice for $\mathcal{K}$ requires a pre-processing step, called *conceptual scaling*,[4] that derives a binary context out of $\mathcal{K}$.

3.2. FCA-based hierarchy analysis

FCA has been successfully applied to class hierarchy design and maintenance[6] with contexts drawn from sets of existing classes by encoding classes and class members as formal objects and attributes, respectively. The resulting conceptual lattice was interpreted as a surrogate hierarchy where formal concepts represent (abstract) software classes and order in the lattice as specialization.

Recently, Huchard *et al.*[7] proposed an extension of the basic FCA mechanisms, called *relational concept analysis* (RCA), that allows links among formal objects to be fed into the lattice construction in order to abstract from those links higher order inter-concept relations similar to UML associations. RCA uses a compound data

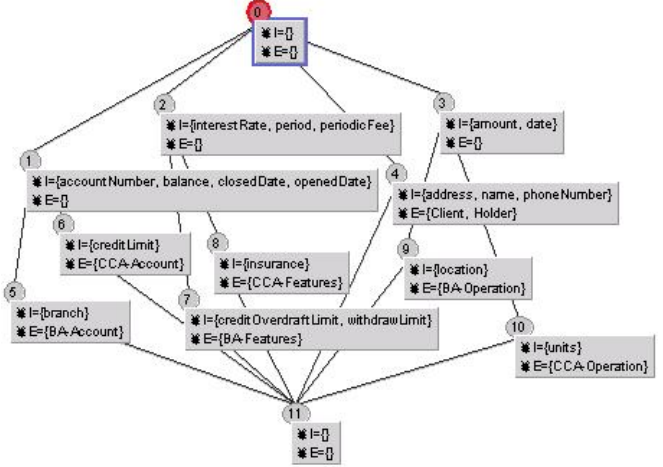|	out-assoc	in-assoc
BA-Account	{Allows, BA-Owns, BA-Has}	
BA-Features		BA-Has
BA-Operation		Allows
CCA-Features		CCA-Has
CCA-Operation		Supports
Client		BA-Owns
CCA-Account	{CCA-Owns, CCA-Has, Supports}	
Holder		CCA-Owns

Figure 2. Left: Relational variables in $\mathcal{K}_{class}$. **Right:** The lattice $\mathcal{L}^0_{class}$.

format, called *relational context family* (RCF), including a set of formal contexts and a set of inter-object relations (translated to *relational* attributes for computational reasons). Each context, called *relational context*, describes a separate sort of domain entities as formal objects, while relations connect entities from (possibly) separate contexts. An encoding of a UML class diagram maps classes and associations to separate contexts of the RCF and the class - association relations to dedicated relational attributes in both contexts (e.g., `in-association`, `target-class`, etc.). An innovative feature of RCA is the inference of relational abstractions connecting formal concepts from links between formal objects, following a tight analogy to the Entity-Relationship data model. In our UML encoding, this means that the intent of a formal concept over the class context may include a relational attribute "pointing at" a formal concept of the associations context, which abstracts from existing links between the corresponding formal objects of class and association type.

The production of inter-concept relations is integrated into the overall lattice construction step by means of *relational scaling* mechanism.[8] RCA uses an iterative lattice construction mechanism that processes all the contexts of a family simultaneously. The construction alternates relational scaling and lattice construction tasks. Thus, each lattice gets more detailed along subsequent iterations (since new concepts are added while the old ones remain) whereas the global construction halts whenever a particular step yields no new concept.

Figure 2, on the left, presents a relational context extracted from the UML diagrams given in Figure 1 in a way that will be presented later on. RCA uses an iterative lattice construction method that processes all the contexts of a family simultaneously by means of *relational scaling* mechanism which encodes relational attributes into binary ones. Figure 3 presents the lattice of the context obtained by concatenating Table 1 and table shown in the left hand-side of Figure 2.

3.3. Naming conflicts resolution

A set of hierarchical elements may overlap either on their *names* or on their *descriptions* (including properties and links to other elements). Both these aspects of an element are to be seen as "proxies" for the denotation behind the element, i.e., the domain entity it represents, which is not necessarily available for a hypothetical automated tool. Indeed, although providing some rigor and a lot of structure, plain UML models, as other hierarchy description languages, are basically natural-language-bound and therefore potentially ambiguous. Obviously, a strong match in both name and description is a good indication for elements representing the same reality. Overlap in only one aspect is trickier case as it may be the sign, except for imprecise modeling, of complex linguistics phenomena intervening such as *synonymous* (same entity, different names, e.g., `Client`/`Holder`) or *polysemous* (same name, different entities, e.g., the `Feature` classes).

These problems have been addressed in the database schema integration field and it is usually suggested that name conflicts be resolved through name normalization based on linguistic resources such as electronic

dictionaries or specialized domain ontologies. Of course, neither is the linguistic-knowledge based approach universally applicable, nor is it possible to solve all conflicts by an automated tool. Therefore, a more realistic agenda for an integration tool could be to detect the suspect areas in the global hierarchy where effective/potential name conflicts reside. FCA offers mechanisms for the detection of typical and exceptional patterns of co-occurrence of formal objects/attributes called implication rules.[9] These can be used to detect discrepancies between descriptions of elements, i.e., mainly classes and associations, having same or close names that may hide a conflict. Structural properties of the lattice and mutual position of those elements within it could be further explored to the same end.

More sophisticated tools could even suggest, for a given pair of elements, the impact of a particular action on them, e.g., merge, creation of a dedicated generalization, split into thinner elements, etc., to the remaining lattice and hence the final hierarchy. Finally, a more extensive and fully manual approach towards name conflict resolution could be based on the *attribute exploration*[10] mechanism which is basically a knowledge acquisition method assisted by a automated tool asking questions to the expert designer. The tool guides the hierarchy designer to the discovery of a minimal set of implications that represent the entire domain while minimizing the number of questions. It is noteworthy that the above mentioned FCA tools such as implication extraction and attribute exploration are yet to be extended to the relational data descriptions that are used here.
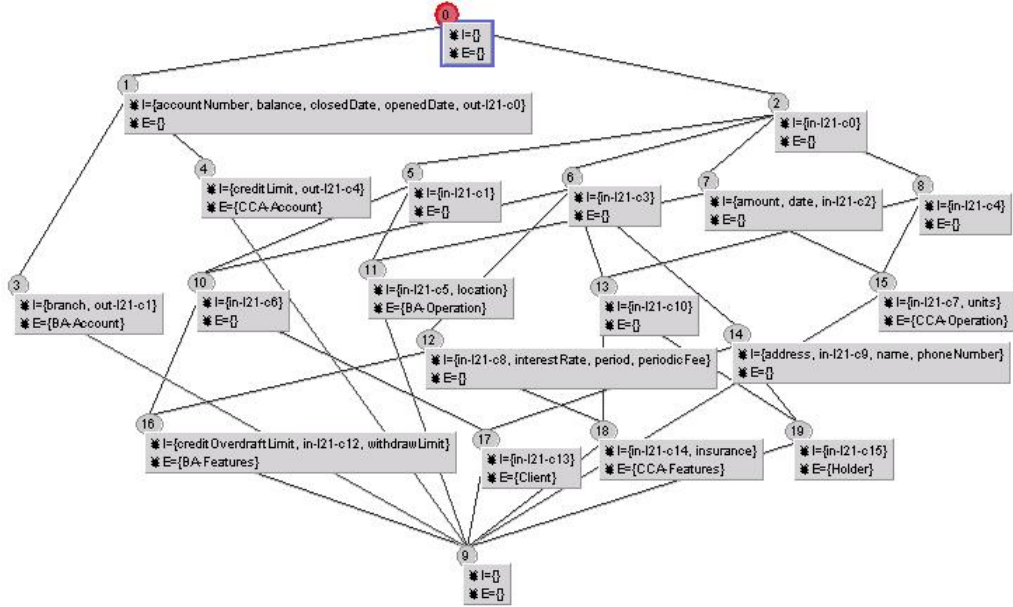

Figure 3. Relational lattice $\mathcal{L}_{class}^1$.

3.4. Description of the approach

The global scenario of integration includes three main stages: *encoding*, *abstraction* and *reverse encoding*. During the encoding stage, the initial hierarchies are transformed into a unique RCF where each context correspond to a sort of objects in the UML meta-model, i.e., classes, associations, methods and variables. Here we limit our considerations to classes and associations. Moreover, the incidences between meta-objects are translated into relational attributes in the RCF, e.g., the attribute `target-class` in Table 2. The encoding of the entire running example is jointly represented by Tables 1 and 2, together with the left-hand side of Figure 2. The resolution of some naming conflicts takes place at this step, in particular conflicts in attribute/method names, which are often less ambiguous than the class or association ones.

At the abstraction stage, a set of inter-related concept hierarchies are constructed on top of the global RCF.[11] The global hierarchy construction starts by processing only the non-relational part of context attributes. Next,

	name	s-mult	t-mult	nav	source-class	target-class
Allows	allows	1	0..N	st	BA-Account	BA-Operation
BA-Owns	ba-owns	1..N	1	st	BA-Account	Client
BA-Has	ba-has	0..N	1	st	BA-Account	BA-Features
CCA-Owns	cca-owns	1..N	1	st	CCA-Account	Holder
CCA-Has	cca-has	0..N	1	st	CCA-Account	CCA-Features
Supports	supports	1	0..N	st	CCA-Account	CCA-Operation

Table 2. Context $\mathcal{K}_{assoc}$ of associations.

the process iterates between construction and scaling of relational attributes in the RCF until a fixed point is reached, i.e., isomorphism between lattices at step $i+1$ and their counterparts at step i. Relational scaling turns each object-valued attribute into a set of binary ones each of whom represents a concept from the conceptual hierarchy of the underlying context. For a relational attribute α that is scaled along an existing lattice on its range context, the resulting scale attributes will have the form α-l#j#i-c#k where j is the context number (here 1 stands for classes and 2 for associations), i refers to the step of the iterative construction process and k is the concept index in $\mathcal{L}_j^i$. For instance, as the values of target-class in $\mathcal{K}_{assoc}$ are objects in $\mathcal{K}_{class}$ (see Table 2), the attributes of the scale context $\mathcal{K}_{target-class}$ correspond to concepts from $\mathcal{L}_{class}^0$ (see Figure 4 on the left). They are used to compute the relational extension of $\mathcal{K}_{assoc}^1$ along the link target-class (see the tc-l10-cX attribute in Figure 4 on the right). Figures 3 and 5 show the final hierarchy derived from the class and association contexts, respectively. Reverse encoding turns the resulting lattices into a global UML class

	tc-l10-c0	tc-l10-c2	tc-l10-c3	tc-l10-c4	tc-l10-c7	tc-l10-c8	tc-l10-c9	tc-l10-c10
BA-Features	X	X			X			
BA-Operation	X		X				X	
CCA-Features	X	X				X		
CCA-Operation	X		X					X
Client	X			X				
Holder	X			X				

	tc-l10-c0	tc-l10-c2	tc-l10-c3	tc-l10-c4	tc-l10-c7	tc-l10-c8	tc-l10-c9	tc-l10-c10
Allows	X		X				X	
BA-Owns	X			X				
BA-Has	X	X			X			
CCA-Owns	X			X				
CCA-Has	X	X				X		
Supports	X		X					X

Figure 4. Left: Scale context $\mathcal{K}_{target-class}$. **Right:** Relational extension of $\mathcal{K}_{assoc}^1$ through $\mathcal{K}_{target-class}$.

diagram (see Figure 6). The process starts with an initial set of class concepts and examines iteratively both lattices following inter-concept relational attributes to figure out the appropriate associations. Initial class set includes all the object-concepts which represent existing local classes and some of the attribute-concepts in $\mathcal{L}_{class}^n$ which means the $\mathcal{GSH}$ of $\mathcal{L}_{class}^n$ is the focus. However, concepts with exclusively relational scale attributes in their intents are left aside at this step as they represent potential classes of high generality level. In our running example, the object-concepts in $\mathcal{L}_{class}^1$ are as follows (corresponding classes in Figure 6 given in brackets): #3 (BA-Account), #16 (BA-Features), #11 (BA-Transaction), #4 (CCA-Account), #18 (CCA-Features), #15 (CCA-Transaction), #17, #19. The concepts #17 and #19 are discarded as they hold only links in their intents. The attribute-concepts in $\mathcal{L}_{class}^1$ are also considered: #1 (Account), #7 (Transaction), #12 (Feature), and #14 (Holder) are kept while the remaining concepts such as #2 and #5 are ignored. The selected concepts, once assigned a name each, constitute a first draft of the global UML model. The next step consists in detecting the appropriate associations from the relational information in the retained class concepts. According to our forward encoding, concepts from $\mathcal{K}_{class}$ have incoming and outgoing links to concepts from $\mathcal{K}_{assoc}$. Moreover, as these links play symmetrical role, tracking one of them is enough. The scale attributes of the incoming link in each selected concept (prefix in) are examined and only those corresponding to a minimum of the related concept set are kept for association computation. For example, the concept #7 in $\mathcal{L}_{class}^1$ which corresponds to the class Transaction has two such attributes in-l21-c0 and in-l21-c2 hence potentially two incoming associations.
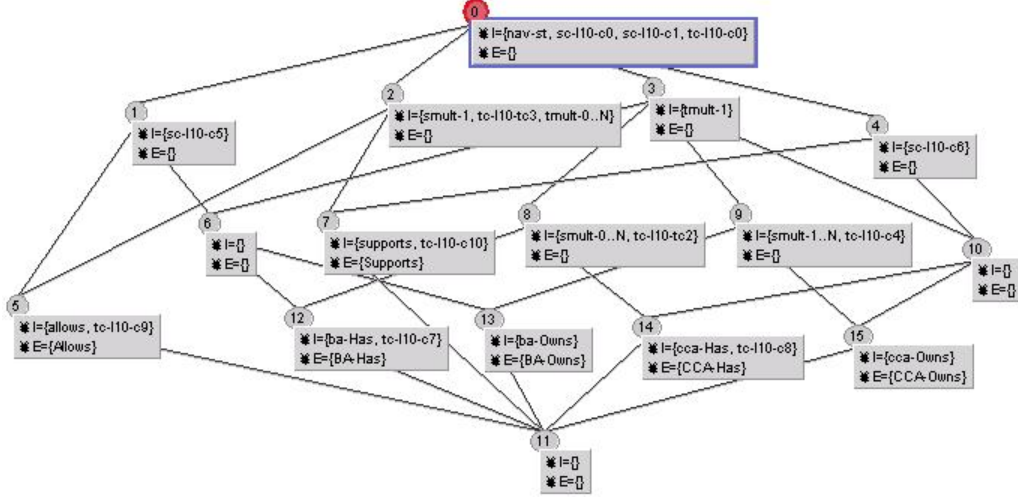
Figure 5. Relational lattice $\mathcal{L}^1_{assoc}$.

However, the concepts #0 (`in-l21-c0`) is a parent of #2 (`in-l21-c2`) in $\mathcal{L}^1_{assoc}$ and thus the unique incoming association for `Transaction` is described by #2. The latter has two source class attributes, `sc-l10-c0` and `sc-l10-c1`, whereby the corresponding concepts in $\mathcal{L}^0_{class}$, i.e., #0 and #1, satisfy $\#1 \leq \#0$. Thus, the source class is #1 which corresponds to the super-concept of the object-concepts for both account classes, i.e., the class `Account` in our interpretation. As the target of the new association is known to be `Transaction`, the exploration of the relations between $\mathcal{L}^1_{class}$ and $\mathcal{L}^1_{assoc}$ continues with another class concept. Figure 6 shows the result of the integration process. Clearly, interesting abstractions have been discovered: `Account` generalizes both sorts of accounts and `Transaction` the respective operations on them, `Holder` become the generic account owner after discarding the `Owner` class as independent one. A new associations have been found as well, i.e., `Supports` which links `Transaction` and `Account`.
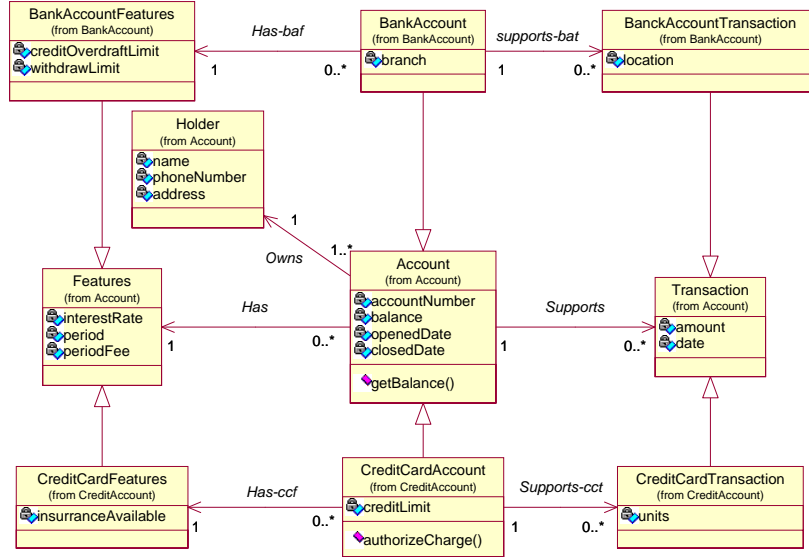

Figure 6. Global UML model.

57

4. RESEARCH AVENUES

We addressed the integration of a set of UML class diagrams into a global one as an instance of the generic hierarchy integration problem. The underlying challenges amount to abstracting new and previously unknown concepts and properly match overlapping parts of the hierarchies. We suggested a RCA-based framework for integration where concepts and relations are organized in a separate but mutually related abstraction hierarchies, the concept lattices. The core abstraction process is highly complex while the gap between a conceptual hierarchy and its encoding in terms of a RCF requires an effective set of tools that facilitate the back and forth navigation.

Our focus in future steps will be on increasing the precision of the mapping from a hierarchy to a RCF and on the design of effective tools for naming conflict detection and on-line adaptive restructuring of the hierarchies. A possible approach would be to assess term "similarity" using a lexical database as in.[12] Moreover, we shall look at interactive simplification (pruning) of the obtained hierarchies and at increasing the automation of the identification of relevant classes and associations as starting point for the reverse-encoding of the global hierarchy. Finally, studying practical cases to assess the quality of the integrated hierarchies.

REFERENCES

1. I. Mirbel and J.-L. Cavarero, "An integration method for design schemas," in *Conference on Advanced Information Systems Engineering*, pp. 457–475, 1996.
2. N. Noy and M. Musen, "Promptdiff: A fixed-point algorithm for comparing ontology versions," in *Proc. 18th AAAI*, 2002.
3. H. Ossher and P. Tarr, "Using multidimensional separation of concerns to (re)shape evolving software," *Commun. ACM* **44**(10), pp. 43–50, 2001.
4. B. Ganter and R. Wille, *Formal Concept Analysis, Mathematical Foundations*, Springer, Berlin, 1999.
5. Object Management Group, Inc., *OMG Unified Modeling Language Specification*, version 1.5 ed., March 2003.
6. 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.
7. M. Huchard, C. Roume, and P. Valtchev, "When concepts point at other concepts: the case of uml diagram reconstruction," in *Proceedings of the 2nd Workshop on Advances in Formal Concept Analysis for Knowledge Discovery in Databases (FCAKDD)*, pp. 32–43, 2002.
8. P. Valtchev, M. H. Rouane, M. Huchard, and C. Roume, "Extracting Formal Concepts out of Relational Data," in *Proceedings of the 4th Intl. Conference Journées de l'Informatique Messine (JIM'03): Knowledge Discovery and Discrete Mathematics, Metz (FR), 3-6 September*, E. SanJuan, A. Berry, A. Sigayret, and A. Napoli, eds., pp. 37–49, INRIA, 2003.
9. J. Guigues and V. Duquenne, "Familles minimales d'implications informatives résultant d'un tableau de données binaires," *Mathématiques et Sciences Humaines* **95**, pp. 5–18, 1986.
10. B. Ganter, "Attribute exploration with background knowledge," **217**, pp. 215–233, 1999.
11. M. Dao, M. Huchard, M. H. Rouane, C. Roume, and P. Valtchev, "Improving generalization level in uml models: Iterative cross generalization in practice," in *Proceedings of the 12th International Conference on Conceptual Structures (ICCS'04)*, **14**, Springer-Verlag, LNCS 3127, July 2004.
12. M. Lafourcade and V.Prince, "Relative synonymy and conceptual vectors," in *Proceedings of NLPRS2001, Tokyo*, pp. 127–134, 2001.

Behavior Consistent Inheritance with UML Statecharts

Markus Stumptner and Michael Schrefl

Advanced Computing Research Centre, University of South Australia,
5095 Mawson Lakes, Adelaide, Australia
{mst|cismis}@cs.unisa.edu.au

ABSTRACT

* Object-oriented design methods express the behavior an object exhibits over time, i.e., the object life cycle, by notations based on Petri nets or state charts. The paper considers the specialization of life cycles via inheritance relationships as a combination of extension and refinement, viewed in the context of UML state machines. Extension corresponds to the addition of states and actions, refinement refers to the decomposition of states into substates. We use the notions of observation consistency and invocation consistency to compare the behavior of object life cycles and present a set of rules to check for behavior consistency of state machines, based on a one-to-one mapping of a meaningful subset of UML 2.0 state machines to Object/Behavior Diagrams.

1. INTRODUCTION

Object-oriented systems organize object types in hierarchies in which subtypes inherit and specialize the structure and the behavior of supertypes. These inheritance hierarchies provide a major aid to the designer in structuring the description of an object-oriented system, and they guide the reader who tries to understand the system by pointing out similarities between object types that are so connected. Informally, specialization means for object life cycles that the object life cycle of a subtype should be a "special case" of the object life cycle of the supertype. There are two ways in which an object life cycle may be made more special. One way is to add new features, which we call *extension*. For example, a "reservation with payment" extends a "reservation" in that it provides for additional features relevant for payment such as billing, paying, and refunding. The other way is to consider inherited features in more detail, which we call *refinement*. For example, a "reservation with alternative payment" refines a "reservation with payment" in that it provides for special means to pay, such as by cash, by cheque, or by credit card.

Extension and refinement should not be employed arbitrarily but according to certain consistency criteria in order to increase understandability and usability. Ebert and Engels[2] pointed out that object life cycles can be compared based upon what a user observes (*observation consistency*) and based upon which actions associated with transitions a user may invoke on an object (*invocation consistency*).

Informally, observation consistent specialization guarantees that if features added at a subtype are ignored and features refined at a subtype are considered unrefined, any processing of an object of the subtype can be *observed* as correct processing from the point of view of the supertype. In our example of "reservation with payment", observation consistency is satisfied if the processing of reservations with payment appears (can be observed) as a correct processing of reservations when all features relevant to payment are ignored.

Weak invocation consistency captures the idea that instances of a subtype can be used the same way as instances of the supertype. For example, if one extends a television set by a video text component, one would expect that the existing controls of the television set should continue to operate in the same way. An extended property, *strong invocation consistency*, guarantees that one can continue to use instances of a subtype the same way as instances of a supertype, even after operations (activities) that have been added at the subtype have been executed. In our television set example, to obey strong invocation consistency means that invoking any video text function should still leave the volume control operative.

---

*An earlier version of this work was published in the Proceedings of the 20th Int'l Conf. on Conceptual Modeling.[1]
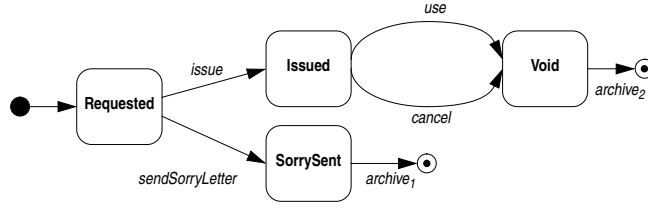
Figure 1. UML statechart diagram of object class RESERVATION

In this paper, we will analyze these properties in the UML statechart context give a set of rules for checking whether an inheritance relationship satisfies them. For space reasons, we have reduced the formalism to a minimum, as detailed definitions are available elsewhere.[1]

UML statechart diagrams are the standard UML "lifecycle" diagrams. Transitions are triggered by call events and their actions invoke the operation at that point.[3] For the sake of brevity we exclude completion and stubbed transitions as well as history states, and assume that an event is associated with every transition. Since transitions are considered instantaneous, we can say that every instance of an object class is at any point in time in one or several states of its state machine. The set of states an object is situated in at a given moment is jointly referred to as *life cycle state*. As given by the UML semantics definition, a transition will be performed if its event occurs and its guard condition is satisfied. For ease of presentation we also assume that at most one transition exists between each pair of states.

Definition: A *UML State Machine* (USM) of a class O consists of a set of states $S \neq \emptyset$ and a set of transitions T of the form $l : (S_1, S_2)$ where $S_1, S_2 \subseteq S$, and $l \in L_O$ with L_O being the set of *event labels*. The labels are of the form e (where e is an event name) if e is associated with one transaction, or of the form e_i (i.e., with a unique index is added), if the event name e occurs multiple times in U_O. There is a distinguished *initial state* α, and a set of distinguished states *final states* Ω. In addition, every transition t can have associated with it a guard condition $(g(t))$.

For a transition $l : (S_1, S_2)$, we call S_1 the set of *source states* of the transition, and S_2 the set of *sink states*.

Since we do not consider action, guard, and activity semantics, the situation of an object at a given point in its lifetime is described by the set of states it occupies in the USM. We refer to this situation as the *life cycle state* of the object. A transition $t = l : (S, S') \in T$ can be performed on a life cycle state σ of an USM U_O, if the source states of t are contained in σ, the event e occurs (where $l = e$ if e is unique in U_O and $l = e_i$ if there are multiple occurrences of e), and the guard $g(t)$ evaluates to True.

EXAMPLE 1. *Figure 1 shows the statechart of object class* RESERVATION.

A statechart diagram of an object class implicitly specifies all legal sequences of life cycle states. A particular sequence of life cycle states of an object class is referred to as *life cycle occurrence* (LCO) of that object class.

EXAMPLE 2. *A possible LCO of object class* RESERVATION *is* [{α}, {requested}, {issued}, {void}] *(cf. Figure 1).*

A life cycle occurrence γ can also be denoted by the sequence of transitions that cause γ, called an *activation sequence*.

EXAMPLE 3. *A possible activation sequence for object class* RESERVATION *is* [request, issue, use, archive$_2$].

Our work in this paper relies on earlier work dealing with Object/Behaviour Diagrams (OBD's).[4] For the purpose of checking behavior consistency, since composite states have no intrinsic semantics, we transform UML statecharts into a *canonical form* such that they contain only transitions between simple states and such that concurrent regions are always explicitly exited. From this state, we can define a homomorphism from USM's to OBD's, called the U2O mapping.[1]
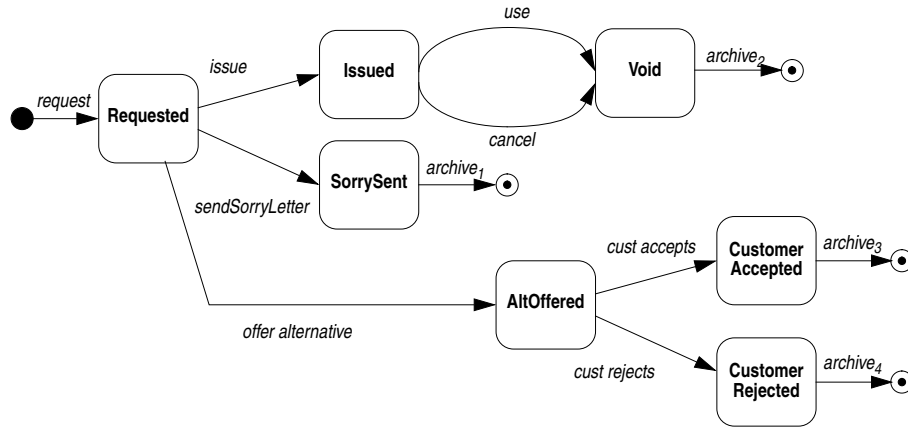
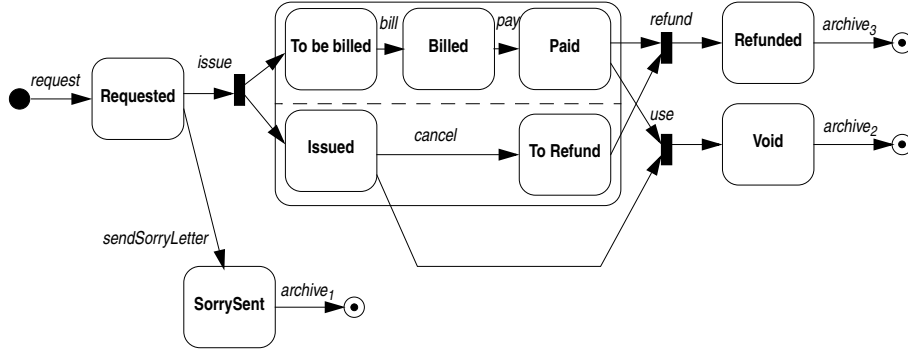**Figure 2.** UML state diagram of object class FRIENDLY_RES

Figure 3. UML state diagram of object class RESERVATION_WITH_PAYMENT

2. CONSISTENT EXTENSION

For UML statechart diagrams, "extension" means adding transitions and states. Simple transitions may become complex transitions and complex transitions may receive additional sink and source states.

EXAMPLE 4. *Figure 2 shows a statechart diagram for object class* FRIENDLY_RES. *This statechart extends the statechart of object class* RESERVATION *shown in Figure 1.*

Adding a "parallel path" is easily achieved in UML if the additional "parallel path" starts in all its alternatives at transitions having the same sink state and, likewise, ends in all its alternatives at transitions having a common source state. Since UML concurrency requirements trivially imply that an object always leaves and enters all concurrent subregions of a composite state simultaneously, other extensions may not be expressible by adding states and transitions to an existing statechart diagram unless behavior consistency is sacrificed. The semantics of the intended extension must then be alternatively expressed by refinement, i.e., the decomposition of states into so-called substates that provide a more detailed description of the inner workings of that state. This special case is described elsewhere,[1] and we ignore it from here on.

2.1. Kinds of Behavior Consistency

Intuitively, one expects the statechart of a subclass to be "consistent" with the statechart of a superclass. In this section, we define precisely what "consistent" means with respect to extension of UML statecharts. Following the example from OBD's,[4] the idea naturally emerges to use the addition of states and transitions in order to define the notion of "consistent extension" of state machines. As an example, consider the statechart diagram FRIENDLY_RES which is based on the diagram for RESERVATION (cf. Figures 2 and 1).

61

Based on our earlier research on Petri net-based behavior diagrams, we found two approaches are common for comparing the behavior of two Petri nets[5]: (1) Abstracting from actions, one can compare the possible sequences of sets of states in which tokens reside and (2) abstracting from states, one can compare the possible sequences in which transitions can be performed. These approaches are usually followed alternatively. In UML, as noted, we compare action sequences, not event sequences, and both approaches coincide since we can denote a life cycle occurrence either by the sequence of its life cycle states or by the activation sequence generating it, whichever is more convenient (see above).

We are now ready for describing the three kinds of behavior consistency[4] in an UML context. As described above, the perspective of **observation consistency** in semantic data models and object-oriented systems is that each instance of a subclass must be observable according to the structure and behavior definition given at the superclass, if features added at the subclass are ignored. A life cycle occurrence of a subclass can be observed at the level of the superclass, if activities, states, and labels added at the subclass are ignored. This is expressed by the notion of the *restriction of a life cycle occurrence* to a supertype, which consists of omitting all transitions and states that do not belong to the supertype from the individual LCS's of the LCO.

EXAMPLE 5. *A possible life cycle occurrence of object class* RESERVATION_WITH_-PAYMENT *(cf. Figure 3) is* [{requested}, {issued,toBeBilled}, {issued,billed}, {issued,paid}, {void}]. *The restriction of this life cycle occurrence to object class* RESERVATION *yields* [{requested}, {issued}, {issued},{issued}, {void}].

Observation consistent extension of behavior requires that each possible life cycle occurrence of a subclass is, if we disregard activities and states added at the subclass, also a life cycle occurrence of the superclass. Observation consistency ensures that all instances of an object class (including those of its subclasses) evolve only according to its statechart diagram. This property is especially important for modeling workflows, where, for example, the current processing state of an order should always be visible at the manager's abstraction level defined by some higher-level object class.

EXAMPLE 6. *The statechart of class* FRIENDLY_RES *depicted in Figure 2 is no observation consistent extension of the statechart of class* RESERVATION *depicted in Figure 1. The restriction of LCO* [{requested}, {altOffered}] *of class* FRIENDLY_RES *to class* RESERVATION *yields* [{requested}, {}], *which is no LCO of class* RESERVATION.

As mentioned above, we distinguish two forms of invocation consistency.

Weak invocation consistency ensures that if an object is extended with new features, e.g., a television set with video text, the object is usable the same way as without the extension, i.e., every activation sequence μ valid on A in U is also valid on supertype A' in U' and all traces of A', when restricted to A's states and transitions, yield a trace for A.

EXAMPLE 7. *The statechart diagram of class* FRIENDLY_RES *depicted in Figure 2 is a weak invocation consistent extension of the statechart diagram of class* RESERVATION *depicted in Figure 1.*

EXAMPLE 8. *The statechart diagram of object class* RESERVATION_WITH_PAY-MENT *depicted in Figure 3 is no weak invocation consistent extension of the statechart diagram of object class* RESERVATION *depicted in Figure 1. The activation sequence [* request, issue, use, archive$_2$ *] is valid for class* RESERVATION, *but not for class* RESERVATION_WITH_PAYMENT, *as* use *cannot be applied on the LCS* {issued,toBeBilled} *reached by executing* [request, issue].

Strong invocation consistency is satisfied if one can use instances of a subclass A' in the same way as instances of the superclass, despite using or having used new operations of the subclass A, i.e., for any activation sequences μ', ν where μ' is of the subtype that consists of actions from $T' \setminus T$ (subtype only), but ν' would be valid for an instance of A, then the concatenated sequence μ', ν' is also valid for A'.

EXAMPLE 9. *The statechart diagram of object class* FRIENDLY_RES *depicted in Figure 2 is no strong invocation consistent extension of the statechart diagram of object class* RESERVATION *depicted in Figure 1: The call event* sendSorryLetter *will perform the transition* sendSorryLetter:({Requested},{SorrySent}) *for every instance of* RESERVATION *in LCS* {requested}, *but the event will be ignored according to UML semantics on an instance of* FRIENDLY_RES *if the call event* offerAlternative *has occurred before and the guard was true.*

2.2. Checking Behavior Consistent Extension in UML 2.0

We rephrase the rules for checking behavior consistency introduced for OBD's[4] in the context of USM's. This rephrasing is valid since the U2O mapping always produces an equivalent OBD, with two exceptions. The mapping ignores guard conditions which are part of the preconditions for UML transitions. Also, the new concept of Protocol State Machines (PrSMs) in UML 2.0 permits the addition of postconditions which have to be considered in the rules. The guard conditions and postconditions therefore need to be explicitly introduced and the corresponding passages are given in italics below. We denote the postcondition for a transition by $p(t)$.

Consider two USM's U' and U of an object class O' and its superclass O, where U' extends the statechart diagram of U by additional states and transitions. Then, the rules to check for behavior consistency of statechart diagram extensions (based on the rules given previously for OBD's[4]) are:

1. The *rule of partial inheritance* specifies that

    (a) the initial states of the statechart diagrams U' and U must be identical

    (b) every transition of U' which is already in U has at least the same source states and sink states than it has in U

    (c) (since the rule of partial inheritance is in the line of covariance) *for each transition t in U' that is already present in U, the guard condition $g'(t)$ in U' must be at least as strong as the guard condition $g(t)$ for t in U: $g'(t) \Rightarrow g(t)$. In a PrSM, the postcondition in U' must be at least as strong as in U: $p'(t) \Rightarrow p(t)$.*

2. The rule of *immediate definition of pre states, post states, and labels* requires that a transition of U may in U' not receive an additional source state or sink state that is already present in U.

3. The rule of *parallel extension* requires that a transition added in U' does not receive a source state or a sink state that was already present in U.

4. The rule of *full inheritance* requires that the set of transitions of U' is a superset of the set of transitions of U.

5. The rule of *alternative extension* requires that

    (a) a transition in U' which is already present in U has in U' at most the source states that the transition has in U.

    (b) (since the rule of alternative extension is considered contra-variant) *for each transition t in U' that is already present in U, the guard condition $g'(t)$ must be in U' at least as weak as the guard condition $g(t)$ in U: $g(t) \Rightarrow g'(t)$. In a PrSM, the postcondition in U' must be at least as strong as in U: $p'(t) \Rightarrow p(t)$.*

Rules 1, 2, and 3 are sufficient to check whether a USM U' is an observation consistent extension of another statechart diagram U. They are also necessary, provided that analogous assumptions to the safety, activity-reduced, and deadlock-free conditions introduced previously for behavior diagrams[4] are taken here.

Rules 1, 2, 4 and 5 are sufficient to check whether a USM U' is a weak invocation consistent extension of another USM U. They are also necessary if conditions corresponding to those introduced for OBD's above are obeyed.

Rules 1 to 5 are sufficient to check whether a USM U' is a strong invocation consistent extension of another USM U.

3. RELATED WORK AND OUTLOOK

UML has introduced an explicit notion of extension of a statechart diagram with the new UML 2.0 standard,[6] (p.493), and the query isConsistentWith() is defined as an operation to check for extension. This compares favorably with the old standard where inheritance of statecharts (referred to as "refinement") was merely discussed briefly in the text). However, the notion of extension is still not restricted in terms of the additions permitted, and although simple states can be replaced by composite states, this only serves the representation of parallelism and not the concept of refinement as described in earlier work.[4]

Other work on inheritance for object life cycles includes work based on state diagrams that are related via graph (homo-)morphisms,[2,7] finite automata,[8] state machines,[9] and statecharts.[10] A discussion of life cycles on Petri net basis, but without completeness or sufficiency results, was given by van der Aalst.[11] Restrictions on inheritance relationships in concurrent object-oriented languages were examined, e.g., by Nierstrasz[12] and America.[13] An approach that expresses subtype relations in terms of implications between pre- and postconditions of individual operations plus additional constraints was given by Liskov and Wing,[14] providing explicit criteria for individual operations, but not for descriptions of complete object life cycles. A more detailed comparison can be found in our previous work.[4]

In this paper we have treated specialization of object life cycles by examining extension and refinement in the context of UML statecharts.

The ubiquity of UML means that despite its shortcomings any step towards capturing the relevant semantic properties of the language is of immense practical relevance. Our work is the first formal scheme for describing consistency of UML lifecycle inheritance in terms of a set of explicit rules and provides a basis for incorporating further aspects of UML. In particular we are applying the same principles to UML 2.0 activity diagrams.

REFERENCES

1. M. Stumptner and M. Schrefl, "Behavior consistent inheritance in UML," in *Proc. Intl. Entity-Relationship Conference (ER 2000)*, (Salt Lake City), Oct. 2000.
2. J. Ebert and G. Engels, "Observable or Invocable Behaviour — You Have to Choose," tech. rep., Universität Koblenz, 1994.
3. Rational Software Corp., *UML Semantics, Version 1.1*, Sept. 1997.
4. M. Schrefl and M. Stumptner, "Behavior consistent specialization of object life cycles," *ACM Transactions on Software Engineering and Methodology* **11**(1), pp. 92–148, 2002.
5. L. Pomello, G. Rozenberg, and C. Simone, *A Survey of Equivalence Notions for Net Based Systems*, LNCS 609, Springer-Verlag, 1992.
6. OMG, *UML Superstructure 2.0 Draft Adopted Specification*, 2004.
7. G. Saake, P. Hartel, R. Jungclaus, R. Wieringa, and R. Feenstra, "Inheritance conditions for object life cycle diagrams," in *Proc. EMISA-Workshop Formale Grundlagen für den Entwurf von Informationsystemen*, U. Lipeck and G. Vossen, eds., *Informatik-Berichte*(3/94), pp. 79–88, (Institut für Informatik, Universität Hannover), 1994.
8. B. Paech and P. Rumpe, "A new concept of refinement used for behaviour modelling with automata," in *Proc. FME'94, Springer LNCS 873*, pp. 154–174, 1994.
9. J. McGregor and D. Dyer, "A note on inheritance and state machines," *ACM SIGSOFT Software Engineering Notes* **18**(4), pp. 61–69, 1993.
10. D. Harel and O. Kupferman, "On object systems and behavior inheritance," *IEEE Transactions on Software Engineering* **28**(9), pp. 889–903, 2002.
11. W. M. P. van der Aalst and T. Basten, "Life-Cycle Inheritance — A Petri-Net-Based Approach," in *Proc. 18th Intl. Conf. on Application and Theory of Petri Nets, LNCS 1248*, pp. 62–81, Springer-Verlag, 1997.
12. O. Nierstrasz, "Regular types for active objects," in *Proc. OOPSLA*, 1993.
13. P. America, "Designing an object-oriented programming language with behavioural subtyping," in *Foundations of Object-Oriented Languages*, J. de Bakker, W. de Roever, and G. Rozenberg, eds., *Springer LNCS 489*, pp. 60–90, 1990.
14. B. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems* **16**, pp. 1811–1841, Nov. 1994.

Domain Modeling in Self Yields Warped Hierarchies

Ellen Van Paesschen - Wolfgang De Meuter - Theo D'Hondt

Programming Technology Laboratory, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

ABSTRACT

Domain modeling can result in a hierarchical set-up in which the modeled entities follow the standard hierarchical taxonomies while the proper execution of the corresponding code demands the reversed hierarchy. Modeling roles and the identity problem are typical cases of these "warped" hierarchies, which are difficult to implement in class-based languages. In the prototype-based language Self, entities are modeled into hierarchies of traits, supporting multiple inheritance, dynamic parent sharing and copy-down techniques. This powerful cocktail of features allows building warped hierarchies in a straightforward and natural manner.

Keywords: Prototypes, multiple inheritance, dynamic delegation, traits, parent sharing, roles

1. INTRODUCTION

Since the advent of Simula, object-oriented languages are promoted as programming languages that facilitate modeling the real world and make it possible to create taxonomies from the entities that surround us. Indeed, many aspects of a problem domain are easily modeled in object-oriented languages: usually, the modeled entities correspond to an object or a class and taxonomies of entities give rise to class-hierarchies. This way of thinking is pretty straightforwardly applied in a prototype-based language like Self as well. The only difference is that one will replace classes and their hierarchies by traits objects and their hierarchies.

Nevertheless, there exists a significant hiatus in this story. During such a modeling process in Self, we experienced a number of occasions where this straight-forwarded approach gives rise to a hierarchical set-up in which the entities follow the standard hierarchical taxonomies but in which the corresponding code demands exactly the reverse version of this hierarchy. We discovered the existence of such *warped hierarchies* while doing *role modeling*, an activity which is known to be far from easy in a class-based language.[4] They also showed up in relation to fundamental and philosophical shortcomings: e.g. a mathematician would consider a circle as a special kind of ellipse, where both axes are equal, while an object-oriented modeler would rather define an ellipse as a descendant of circle.

Warped hierarchies cannot be implemented in class-based languages. However, this is perfectly feasible in Self, thanks to multiple inheritance, parent sharing and copy-down techniques. We will illustrate this using the circles/ellipses example and the role modeling case.

2. PROTOTYPE-BASED LANGUAGES

2.1. In General

In general, prototype-based languages (PBLs) can be considered object-oriented languages without classes. The most interesting feautures of a PBL are *creation ex nihilo*, *cloning*, *dynamic inheritance modification*, *delegation with late binding of self*, *dynamic parent modification*, and *traits objects**. Many PBLs have been designed in research labs. Examples are Self,[7] Agora,[2] Kevo[9] and NewtonScript.[8] A taxonomy can be found in.[3] We will elaborate on the PBL Self, since it is a textbook example of a PBL and moreover, includes a mature programming environment.

Send e-mail correspondence to {`evpaessc,wdmeuter,tjdhondt`}`@vub.ac.be`

*To avoid copying behavior every time an object is cloned, the SELF-group[11] introduced **traits objects**: storing the shared behavior in an object and let the cloned objects inherit from it, i.e. a kind of class-based programming in a PBL

2.2. Self

Self is closely related to the syntax and semantics of Smalltalk[5] but Self has no classes. Objects in Self are *created ex-nihilo* by putting slot names (together with a possible initial filler value for that slot) between vertical bars, separated by dots. The following code, for example, creates an ex-nihilo myPoint[†] object:

```
myPoint: (|parent* = traits clonable. x <- 3. y <- 4.
          addPoint: point = ((copy x: x + point x)
          y: y + point y)|)
```

Self visualizes its objects with outliners, cfr. figure 1. A slot marked with an asterisk is a parent slot and makes
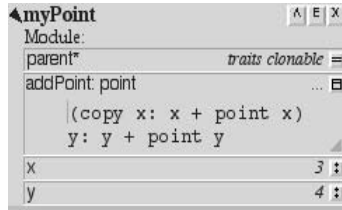

Figure 1. The self-contained myPoint object combines data and behavior

the child inherit all the slots of the parent slot. In this way, myPoint inherits (its behavior) from the *traits object* clonable[‡], and has two data slots containing an x and a y coordinate. The remaining method slot contains a method for adding two points, by *cloning* point and initialize it with the added x and y coordinates.

Self implements a *delegation* mechanism that respects the *late binding of* self. Next to *dynamic inheritance and parent modification*, this delegation mechanism also supports *parent sharing*, i.e. when two or more child objects share the same parent object. This kind of sharing is typical for all PBLs. *Child sharing* (multiple inheritance), on the other hand, when two or more parent objects share the same child object, is a specific feature of Self. When modeling knowledge these two inheritance features are constantly combined.

3. MULTIPLE INHERITANCE IN SELF

When modeling a data type in Self, the data (specific for each "instance" of this data type) is contained in a prototype while the behavior (shared by all objects of this data type) is typically gathered in a traits object. All prototypes inherit their behavior from the traits object, which in his turn often inherits from traits clonable:

```
traits myPoint = (|parent* = traits clonable.
               addPoint: point = ((copy x: x + point x)
               y: y + point y)|)

myPoint = (|parent* = traits myPoint. x .  y|)
```

The graphical representation is illustrated in figure 2. To obtain *a* point, we clone the myPoint prototype and set the x and y coordinates.

```
(myPoint copy x: 1) y: 2.
(myPoint copy x: 3) y: 4.
```

[†]We use the name myPoint since Self already implements a point object

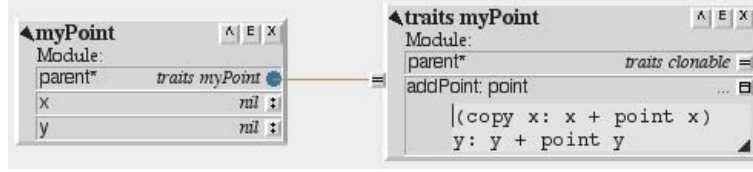[‡]Most concrete not-unique objects in the SELF world are descendants of the top-level traits object traits clonable.

Figure 2. The `myPoint` prototype inherits its behavior from `traits myPoint`

Both points now share the `traits myPoint` object since they both contain a copy of the `parent*` pointer of the prototypical `myPoint`, i.e. the most common form of parent sharing.

When we want to create for example a *coloured* point, data and behavior are to be inherited from a normal point. First, a prototypical `coloured point` is created that inherits its behavior from a corresponding `traits coloured point` object. Naturally, the `traits coloured point` inherit behavior from the `traits myPoint`, since the behavior of a coloured point will be a specialization of a normal point's behavior. On the other hand, the `coloured point` prototype can inherit the coordinates of the normal `myPoint`, and extend them with an extra slot to contain the colour, see figure 3. Remark that this multiple inheritance structure is a diamond. Imagine a
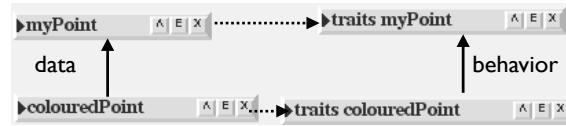

Figure 3. `colouredPoint` inherits data and behavior from `MyPoint`

method `m` in `traits myPoint` that is overridden in `traits coloured point`. When we now send the message `m` to a `coloured point` we get a *name collision*: the method lookup algorithm finds `m` in `traits coloured point` (overriding method) but also in `traits myPoint` (original method) via the data inheritance link with `myPoint`. The early version of Self solved this ambiguity with obscure language mechanisms like *prioritized* parents or the *tie-breaker sender path* rule, which proved to be rather unsatisfying. In the current version of Self we have to resolve ambiguous methods manually by adding a *directed resend* in `coloured Point`. Calling `m = (traits colouredPoint.m)` would invoke the overridden method while `m = (traits myPoint.m)` would return the original method. But then we violate the principle of traits-based inheritance, since we add shared behavior in a prototype in stead of into the corresponding traits object.

Self avoids this problem by performing a `copy-down` of the `myPoint` prototype: this mechanism for data inheritance copies (some of) the slots of the receiver into a new object, ensuring that changes (adding/removing slots) to the receiver are propagated to all copied-down children. Next, we override the `parent*` pointer with the `traits colouredPoint` object. In this way, `colouredPoint` inherits all the data of `point` except for its parent: this implies that there are no name collision when `traits coloured point` override methods of `traits myPoint` In fact, `copy-down` allows a kind of class-based programming: copy-down can be considered as creating a subclass. The `colouredPoint` and `myPoint` inheritance structure is illustrated in figure 4. The complete Self code for the literal point objects can be found in Appendix A.

4. WARPED SELF INHERITANCE HIERARCHIES

It is our experience that modeling domains in Self often results in a rather classical object organisation, differing little from a class-based set-up. However, we found two examples where the transition from domain model notation to code notation gives rise to warped inheritance hierarchies, namely the *identity problem* of circles and ellipses and *role modeling.*
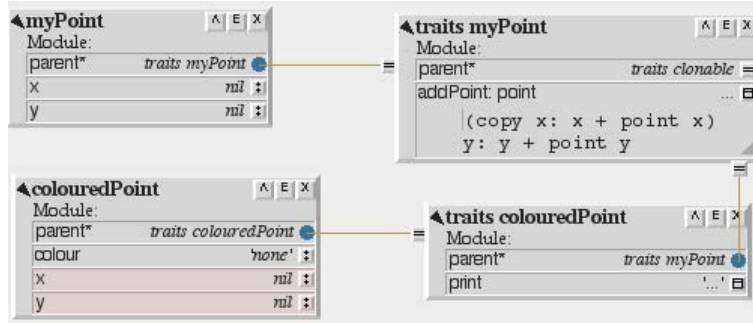
Figure 4. `colouredPoint` inherits data from `myPoint`, `traits colouredPoint` inherits behavior from `traits myPoint`

4.1. Is a circle an ellipse?

Although not many OO-programmers are aware of it, from the *real world* (domain model) point-of-view, a circle really *is-a* kind of ellipse (with major semi-axis a = minor semi-axis b = radius) and thus the code should see circles as specializations of ellipses. In a class-based language the circle type *can* be implemented as a subclass of the ellipse type, resulting in inefficient code since circle will not use all instance variables inherited from ellipse. The difficulty is mainly caused by the fact that the data of circle is less specialized than ellipse's data while the behavior of circle is more specialized than ellipse's behavior. An extra problem in this context, is that circles can receive messages intended for ellipses, transforming them dynamically into ellipses, and vice versa. E.g. when a circle receives a `stretch` message that largens the width of an ellipse: a circle would become an ellipse but be of class "Circle"!

Thanks to the separation of data and behavior inheritance, and dynamic modification of parents, Self allows us to model the identity example with warped hierarchies. We let `ellipse` inherit data from `circle` (since it extends it with an extra slot for a major semi-axis value), while `traits circle` inherit from `traits ellipse`, see figure 5. As mentioned in the previous section, the diamond set-up can be broken by defining `ellipse` as
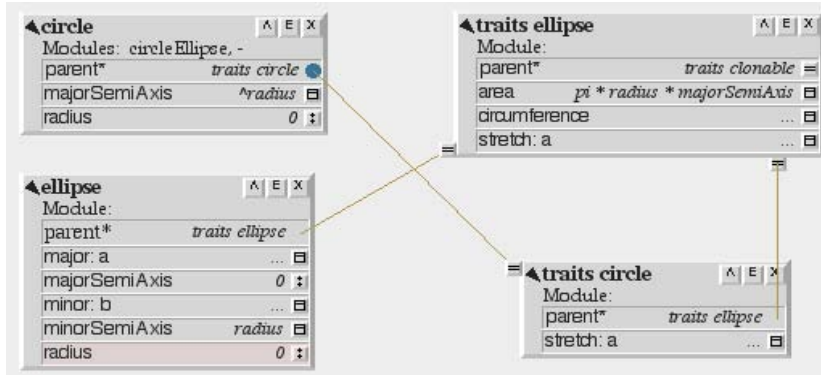
Figure 5. Warped hierarchy of `circle` and `ellipse`

a `copy-down` of `circle` and assigning the `parent*` pointer to `traits ellipse`. Thanks to the late binding of the `self` variable, the correct data is accessed when executing methods (e.g. `area`, `circumference`) - and thus polymorphism is ensured. When a circle is stretched to an ellipse, we add all the slots of the `ellipse` prototype into the circle, thereby overriding the `parent*` pointer from `traits circle` to `traits ellipse`. Vice versa, an ellipse whose major semi-axis is stretched to the same value as its minor semi-axis, becomes a circle, by removing all slots that were not copied-down from circle and by overriding the `parent*` pointer from `traits ellipse` to `traits circle`. In this way, objects seem to change the prototypes they were cloned from dynamically.

4.2. Role Modeling

The roles a person can perform are on one hand subtypes of a person: e.g. an engineer *is-a* kind of person. On the other hand, when a role type inherits from person, how will we - in a class-based language - model that this person can perform other roles? E.g. when both engineer and manager are subclassed from person and we want to model a person that is both manager and engineer. When we instantiate the manager class, the engineer class will be invisible and vice versa. Creating combination classes is not feasible: imagine the difficulties when a person can change dynamically between a large set of roles[4]! Alternatively, roles are often modeled with aggregation: a set of roles is held by an instance variable in the person class. By delegating the messages of person to its roles, polymorphism is simulated.[4]

The real difference with the previous example lies in the fact that roles can be added or removed *dynamically*, and that a person can have *multiple* roles implementing the *same* method. Simply warping the data hierarchy between a person and its roles will not be sufficient.

Therefore, we implemented `receiver createDataparent:parent`[§] as a reverse of the `copy-down` method: in stead of copying down the data from the receiver into a new child object, the data of the parent is copied down into the receiver. *We now create dynamically parents in stead of children.* Due to the dynamic character of the derived types, we also provided a `receiver remove Dataparent:parent` that removes all copied-down data from the receiver.

Consider a `person` prototype that inherits from `traits person`, and a set of role prototypes (e.g. `manager`, `engineer`) inheriting from their traits (e.g. `traits manager`, `traits engineer`), that in their turn all inherit from `traits person`. A `person` that dynamically starts performing a role is implemented by dynamically adding this role's prototype as a data parent to the `person` prototype. Next, we remove[¶] the person's `parent*` link to `traits person` since these are already inherited via the role data parent. Due to the multiple inheritance in Self we can add as many roles as we like, cfr. figure 6. When a `person` dynamically stops performing a role, we
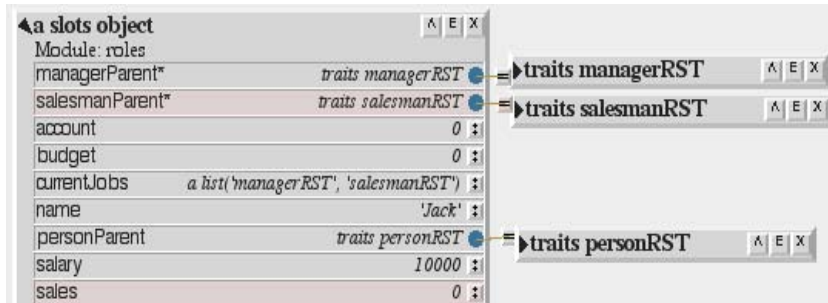

Figure 6. Warped hierarchy of `person` and two roles

remove the data parent. When there are no more role data parent we make the `traits person` visible again. In fact, the *desired behavior is added or removed dynamically.*

To ensure polymorphism we need to intercept the dynamic diamond that is implemented by a person that inherits from two role data parents whose traits both inherit from `traits person`. More specifically, when two roles of a person both override the same method in their traits, sending the corresponding message to person will cause a VM ambiguity error. Our approach depends on the way the methods should be combined from the view point of person. E.g. when we send the message `pay` to `person`, he should get payed for *all* the roles he performs . Therefore, we implemented `delegateMethod:selector` that *sequentially resends the message to all the data parents*, i.e. the roles, of person. However, it is possible that we only want to invoke a specific method, defined in the role in whose context we currently see the person. E.g. when we send the message `lunch` to `person`, she

---

[§]Meta-programming methods heavily use the technique of Self mirrors: an object is reflected on by means of a mirror; manipulating the mirror results in manipulating the object

[¶]We simply make a plain slot from this parent slot

might simulate the specific behavior to have lunch with her best friend and not, for example, with her boss and some clients of the company she works for. In that case, we suggest to turn on/off the parent visibility of the desired behavior, i.e. (temporarily) changing the parent slots, that point to the traits of currently non-desired roles, to normal slots. In this way we maintain the illusion that we are dealing with one person performing various roles.

5. CONCLUSION

PBLs, especially Self, are a suitable medium for modeling knowledge, with powerful inheritance mechanisms which outrank the class-based ones. We experienced the phenomena of warped hierarchies and implemented a technique, that profits from the separated data and behavior inheritance in Self, and intercepts the dangers of multiple inheritance in this context. We have the "gut feeling" that these warped hierarchies are one of the fundamental "missing links" in the transformation process that leads domain models to code.

REFERENCES

1. G. Blashek, *Object-Oriented Programming with Prototypes.*, Springer Verlag, 1994
2. W. De Meuter, *Agora: The Scheme of Object-Orientation, or, the Simplest MOP in the World.* In J. Noble, A. Taivalsaari, I. Moore, eds.: *Prototype-based Programming: Concepts, Languages and Applications*, 1998
3. C. Dony, J. Malenfant, D. Bardou, "Classifying Prototype-based Programming Languages.", In J. Noble, A. Taivalsaari, I. Moore, eds.: *Prototype-based Programming: Concepts, Languages and Applications*, 1998
4. M. Fowler, *Dealing with Roles.*, Collected papers from the PLoP '97 and EuroPLoP '97 Conference, Technical Report wucs-97-34, Washington University Department of Computer Science; http://www2.awl.com/cseng/titles/0-201-89542-0/apsupp/roles2-1.html, 1997
5. A. Goldberg, D. Robson, *Smalltalk-80: The Language and Its Implementation.*, Addison-Wesley, 1983
6. H. Lieberman, "Using prototypical objects to implement shared behavior in object oriented systems.", In N. Meyrowitz, ed.: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).*, Volume 22, 214 - 223, 1987
7. R. Smith, D., Ungar, "Programming as an Experience: The inspiration for Self.", In J. Noble, A. Taivalsaari, I. Moore, eds.: *Prototype-based Programming: Concepts, Languages and Applications*, 1998
8. W. Smith, "NewtonScript: Prototypes on the Palm.", In J. Noble, A. Taivalsaari, I. Moore, eds.: *Prototype-based Programming: Concepts, Languages and Applications*, 1998
9. A. Taivalsaari, *A Critical View of Inheritance and Reusability in Object-oriented Programming.* PhD thesis, University of Jyvaskyla, Finland, 1993
10. D. Ungar, R. Smith, *Self: The Power of Simplicity.*, In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Volume 22, ACM Press, 1987
11. D. Ungar, C. Chambers, B. Chang, U. Holzle, *Organizing programs without classes.*, Lisp and Symbolic Computation 4, 223 - 242, 1991
12. *Self Home Page*, http://research.sun.com/research/self/.

APPENDIX A. SELF CODE

A.1. myPoint objects

```
globals _AddSlots: (|myPoint|).
traits _AddSlots:(|myPoint|).

myPoint: (|parent* = traits clonable. x <- 3. y <- 4.
          addPoint: point = ((copy x: x + point x)
          y: y + point y)|).

traits myPoint: (|parent* = traits clonable.
                addPoint: point = ((copy x: x + point x)
```

```
            y: y + point y)|).

myPoint: (|parent* = traits myPoint. x .   y|).
```

A.2. colouredPoint objects

```
globals _AddSlots: (|colouredPoint|).
traits _AddSlots:(|colouredPoint|).

traits colouredPoint: (|parent* = traits myPoint.
                        print = ('...')|).

colouredPoint: (((myPoint _Mirror) createSubclass) reflectee)
                      _AddSlots: (|parent* = traits colouredPoint.
                                     colour <- 'none'|).
```

Inheritance Decoupled:
It's More Than Just Specialization

L. Robert Varney and D. Stott Parker

University of California at Los Angeles

ABSTRACT

The predominant design of object-oriented programming languages provides insufficient support for interface abstraction and implementation inheritance, spreading implementation bias and impairing evolution. While some ascribe such problems to inheritance and propose restraining or eliminating it, we trace the origin of implementation bias to concrete instantiation dependencies, and propose interface-oriented programming (IOP) as a solution. IOP decouples the inheritance mechanism from implementation binding, and provides an interface-oriented form of inheritance that keeps implementation bias in check and is useful for both specialization and adaptation.

1. INTRODUCTION

The implementation of inheritance has been problematic. Multiple inheritance has fallen out of favor, and some recommend limiting inheritance to specialization,[1] or replacing it entirely with composition.[2] While understandable, these viewpoints blame a useful concept for troubles caused by particular implementations. Inheritance, especially multiple inheritance, is a powerful tool for incremental development, useful both for subtyping (specialization) and subclassing (adaptation). The challenge is to implement it effectively, and this position paper proposes a new way of thinking about inheritance and a new approach to its implementation.

The motivation for this work is not to refute these views but to solve specific practical problems: excess implementation bias and poor support for incremental programming. Implementation bias is overdependence on particular implementations, caused by poor interface abstraction whenever objects are explicitly or implicitly instantiated in terms of class name references. These concrete instantiation dependencies in turn weaken incremental programming mechanisms such as composition and inheritance by tightly coupling conceptually separable concerns.

Our proposed solution is a new approach to language design and software engineering we refer to as interface-oriented programming (IOP).[3] IOP eliminates implementation bias by strictly separating interfaces from implementations and decoupling the inheritance mechanism from implementation binding. IOP also encourages new forms of incremental programming that resolve traditional complications of inheritance and enable new levels of automation.

The rest of the paper is organized as follows. Section 2 describes the problems of implementation bias and non-incremental programming and outlines our solution, and section 3 provides an example. The status of this work, and questions for discussion at the workshop are presented in section 4. Related work is highlighted at appropriate points throughout the paper.

2. PROBLEMS AND A SOLUTION APPROACH

Existing proposals for controlling implementation bias move implementation dependencies around instead of eliminating them. In particular, abstract factories[4] replace direct instantiation dependencies with indirect ones, yielding dependencies on factory implementations. Similarly, parameterized components[5] and mixins[6][7][8] introduce parameters for abstract base classes or contained objects, but shift the responsibility to correctly instantiate these objects to the supplier of the actual arguments.

Authors' e-mail: {varney,stott}@cs.ucla.edu

73

Available methods for incremental programming are also insufficient. The nesting of collaboration-oriented designs[9][10][11] forces designers to co-locate specifications of independent concepts, whereas separately specified partial elements[12][13] must somehow be recombined to create consistent wholes. Some combination methods require that factors be composed manually and then checked for consistency, and other methods such as aspect-oriented programming provide automated support for composition but without any guarantees about what the compositions mean.[14]

As a result of these limitations, we are faced with an unfortunate tradeoff in component design: encapsulate implementation choices and accept inflexibility, or expose them in interface parameters and allow their influence to spread. We contend that the root of the dilemma is strong coupling of interfaces with implementations, coupling that impairs the usefulness of incremental programming techniques such as composition and inheritance. Although others (e.g., Snyder[15]) have called for the separate treatment of interface and implementation, and for the separate handling of subtyping and subclassing, no language we know of separates these issues sufficiently to resolve the dilemma. For example, the separate treatment of abstract interfaces as first-class entities in Java and C# is tenuous, as these languages effectively merge interface and class hierarchies and force programmers to identify concrete implementations by name in places where abstract interfaces would suffice conceptually.

An effective and scalable solution requires all program dependencies to be strictly interface-oriented, a seemingly radical practice that is impossible given the languages of today. Abstract interfaces should be mandatory and ubiquitous, not optional constructs used occasionally. To realize this principle demands that incremental programming mechanisms such as inheritance and composition be decoupled from implementation binding, and that concern separation mechanisms be complemented by facilities for automatic and consistent integration of concerns, leading to a novel form of object-orientation we call interface-oriented programming (IOP).

3. INTERFACE-ORIENTED PROGRAMMING BY EXAMPLE

In this section we introduce selected elements of interface-oriented programming through an example in a language we are developing called ARC (Abstractions, Representations, and Contexts). At the statement level ARC is similar to Java, but it replaces Java's interfaces, classes, and packages with ARC's abstractions, representations, and contexts.* Only abstractions and representations will be considered further here. All types in ARC are declared using abstraction names only – representation names are never used. A client that dynamically instantiates an object does so using an abstraction, and a representation that inherits some base implementation also does so using an abstraction. Thus, clients and inheritors never know what representations they are using.

To illustrate IOP we will now describe the design of a stack in ARC. Figure 1 shows the abstractions `Stack<T>` and `List<T>`, defined as subtypes of `Collection<T>`, along with three partial representations of `Stack<T>`, `RSc`, `RSi`, and `RSall`.

Representation `RSc` implements the basic methods of a stack in terms of an aggregated list object, and assumes that the bulk operation `pushAll` is provided by some other partial representation. Furthermore, the method constraint in `RSc` constrains the implementation of `pushAll` to be defined in terms of `push` (this is similar to the information included in Lamping's specialization interfaces[16]). Since we are using composition here, we must explicitly forward method calls to the appropriate methods of the list object. The list object is appropriately encapsulated in `RSc`, preventing clients from misusing it.

In contrast, representation `RSi` uses interface-oriented implementation inheritance of `List<T>` to accomplish the same effect more efficiently, both in terms of run-time overhead and notationally. Despite the fact that the `List<T>` base is invisible to clients, no forwarding or adaptation is needed for the `empty` and `contains` methods, as the versions provided by `List<T>` are unified by assumption with those required by `Stack<T>` (because they come from an assumed common base abstraction, `Collection<T>`). Forwarding is needed to adapt the other methods of `Stack<T>` to `List<T>`, however. This forwarding is needed due to the translation of one interface to another, but it is still done in terms of self calls. Thus, this example illustrates use of inheritance for both specialization and adaptation.

---

*It is debatable whether ARC's differences warrant these name changes. But there are nontrivial differences, in particular between ARC representations and Java classes, an we conservatively chose to change the names in order to avoid confusion.

```
abs Collection<T> {                              rep RSc represents Stack<T> assumes Stack<T> {
  boolean contains(T);                             List<T> list;
  boolean empty(T);                                Stack<T>() { list = new List<T>(); }
  control Iterator<T> elements();                  boolean empty() { return list.empty(); }
}                                                  boolean contains(T x) { return list.contains(x); }
                                                   void push(T x) { list.addRear(x); }
abs List<T> extends Collection<T> {                T top() { return list.rear(); }
  List<T>();                                       T pop() { return list.removeRear(); }
  void addFront(T);                                void pushAll(Collection<T> c) uses push(T);
  void addRear(T);                                }
  T removeFront();
  T removeRear();                                rep RSi represents Stack<T> assumes Stack<T>, List<T> {
  T front();                                       Stack<T>() assumes List<T>() { }
  T rear();                                        void push(T x) { addRear(x); }
}                                                  T top() { return rear(); }
                                                   T pop() { return removeRear(); }
abs Stack<T> extends Collection<T> {               void pushAll(Collection<T> c) uses push(T);
  Stack<T>();                                     }
  void push(T);
  void pushAll(Collection<T> c);                 rep RSall represents Stack<T> assumes Stack<T> {
  T pop();                                         void pushAll(Collection<T> c) uses push(T) {
  T top();                                           for (T x in c.elements()) push(x);
}                                                  }
                                                 }
```

Figure 1. ARC Abstractions and Representations for a Stack

The effect of an `assumes` clause in ARC is similar to that of an `extends` clause in Java – they both may involve extension or overriding of some base implementation. However, an assumed base in ARC is an unspecified representation of a specified abstraction, not an explicitly named implementation class.

Finally, representation `RSall` implements the bulk operation `pushAll` in terms of `push`. Since each representation only partially implements the `Stack<T>` interface, each must also assume `Stack<T>` as a basis. Multiple representation fragments must therefore be combined to yield a whole `Stack<T>`, and these in turn must be combined with some suitable representation of `List<T>` (not shown). We can now represent `Stack<T>` in two general ways:

- `RSc` ⊕ `RSall` ⊕ `rep(List<T>)`

- `RSi` ⊕ `RSall` ⊕ `rep(List<T>)`

where ⊕ indicates representation unification and the term `rep(List<T>)` requires further expansion, generating even more possibilities when alternative ways of representing `List<T>` are substituted for it. `RSall` is combinable with both `RSc` and `RSi` because its method constraints and base assumptions are unifiable. If `RSall` had implemented `pushAll` in terms of something other than `push` it would not be consistent with `RSi` or `RSc`.

Our representations of `Stack<T>` do not and should not care about which of these representations `List<T>` are chosen, and similarly, the client of `Stack<T>` should not care which of these representation combinations is used to represent `Stack<T>`. Thus, when a client invokes:

    Stack<T> s = new Stack<T>();

the instantiation is interpreted non-deterministically – all the client expects is that some valid representation of `Stack<T>` will be used. Not only is the client freed from the burden of selecting a representation of `Stack<T>`, the client can also ignore the selection of representations for other subordinate abstractions. This is in stark contrast to existing approaches (such as mixins or parameterized components) that burden the client with the need to select and compose a mutually consistent set of implementation fragments. Our approach for handling this automatically relies on two new mechanisms: representation inference and representation selection.

Representation inference can be thought of as a new form of program linking, a step that occurs after compilation and before run-time, generating for each interface a set of alternative complete representations based on available partial representations. Partial representations must be unified in a way that provides a

complete implementation of all interface methods and also respects each partial representation's local constraints on assumed base types and unspecified methods. Representation selection is simply the means to select one of the available alternatives. This mechanism amounts to a built-in factory mechanism and can be as simple or as sophisticated as desired. We envision a number of approaches that are configurable and controllable at the meta-level, from simple arbitrary selection of one of the available alternatives for each interface considered separately, to context-dependent, evolutionary optimization of the overall set of representation choices for a whole application.

4. SUMMARY AND DISCUSSION

In summary, IOP forces all program interdependencies to be expressed in terms of abstract interfaces, separating the client of an interface from its implementation. Incremental specialization and adaptation also occur using interface-oriented inheritance, separating derived implementations from their foundations. Partial implementation of interfaces is allowed, separating independent segments of an implementation from each other, and encouraging encapsulation of a minimal set of design decisions in each component. This provides a form of mixin-inheritance that separates subtyping from subclassing, but supports constructors and does not require explicit composition of incremental implementation fragments.

Consistent assembly of complete components from partial ones relies on interface-oriented constraints defined locally in terms of assumed base types and unspecified methods. But rather than use such constraints to check the consistency of manually crafted compositions, IOP applies these constraints in a new form of program linking called representation inference to automatically generate a set of candidate component assemblies from the available parts, avoiding the fragile base class problem[17] without imposing conservative constraints on inheritance, and without exposing "specialization information" through client interfaces. Whole implementations of abstract interfaces are then selected automatically from among the inferred alternatives using a user-controllable representation selection mechanism.

At this point the detailed design of the ARC language and the implementation of the compiler and supporting mechanisms are in progress. The basic questions to be addressed by this work include:

1. How should local representation constraints be expressed and used to synthesize complete representations?

2. What is the relationship between IOP and formal approaches to specification and verification – can model-based specification approaches be extended to IOP?

3. A segmented object model is required – what are its costs?

4. How should representation selection be controlled?

Perhaps the most striking result is that a strictly interface-oriented approach forces programmers to delegate to some other agent in the system the responsibility for two tasks normally in their purview, the tasks of (1) composing complete implementations from incrementally defined ones, and (2) selecting suitable complete implementations of an interface from among several alternatives.

In the end, when combined with support for representation inference and evolutionary representation selection, IOP can radically alter our approach to software development. Rather than craft systems to occupy single, fixed points in some solution space, we can instead describe the system by articulating the space itself, and defer to a separate agent the task of growing and adjusting the system in response to changes in the system's environment.

REFERENCES

1. M. Torgersen, "Inheritance is specialisation," in *Proceedings of the ECOOP Inheritance Workshop*, 2002.
2. P. Frölich, "Inheritance decomposed," in *Proceedings of the ECOOP Inheritance Workshop*, 2002.
3. L. R. Varney, "Interface-oriented programming," Technical Report TR-040016, UCLA Department of Computer Science, 2004.

4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

5. D. Batory, J. Liu, and J. N. Sarvela, "Refinements and multi-dimensional separation of concerns," in *Proceedings of SIGSOFT ESEC/FSE*, pp. 48–57, 2003.

6. G. Bracha and W. Cook, "Mixin-based inheritance," in *Proceedings of ECOOP/OOPSLA*, 1990.

7. M. Flatt, S. Krishnamurthi, and M. Felleisen, "Classes and mixins," in *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pp. 171–183, 1998.

8. D. Ancona, G. Lagorio, and E. Zucca, "Jam: A smooth extension of java with mixins," in *Proceedings of ECOOP*, pp. 154–178, 2000.

9. E. Ernst, "Family polymorhpism," in *Proceedings of ECOOP*, pp. 303–326, 2001.

10. M. Mezini and K. Ostermann, "Conquering aspects with caesar," in *Proceedings of AOSD*, 2003.

11. M. Veit and S. Herrmann, "Model-view-controller and object teams: A perfect match of paradigms," in *Proceedings of Aspect-Oriented Software Development*, 2003.

12. C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *Proceedings of ECOOP*, pp. 419–443, 1997.

13. N. Schärle, S. Ducasse, O. Nierstrasz, and A. Black, "Traits: Composable units of behavior," in *Proceedings of ECOOP (to appear)*, 2003.

14. J. Aldrich, "Open modules: Reconciling extensibility and information hiding," in *Proceedings of Workshop on Software Engineering Properties for Languages for Aspect Technologies (SPLAT 04)*, 2004.

15. A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *Proceedings of OOPSLA*, pp. 38–45, 1986.

16. J. Lamping, "Typing the specialization interface," in *Proceedings of OOPSLA*, pp. 201–214, 1993.

17. L. Mikhajlov and E. Sekerinski, "A study of the fragile base class problem," in *Proceedings of ECOOP*, 1998.