

Generalized Object-Oriented Modeling of Behaviour: Behaviour = Records + Meta-Information + Algorithms A Research Agenda

Christos KK Loverdos and Georgios Gyftodimos

Department of Informatics & Telecommunications, University of Athens
{loverdos,geogyf}@di.uoa.gr

Abstract. We use the fundamental object-oriented notion of *record* and enhance it with *meta-information* at various levels (the record level and the attribute/method level) in order to build behavioural models as extensions or generalizations of object-oriented programming. With these ingredients as building blocks, we are able to produce notions such as inheritance of any kind by handling meta-information with the use of *algorithms*. We are particularly interested in the dynamic aspects of the framework which are defined at two distinct levels: the dynamic definition of behavioural models through *meta-information* and their dynamic execution by using suitable pluggable *algorithms*.

Our primary goal is to arrive at a unified framework for the definition and implementation of object-oriented behavioural models. Our on-going research should lead to — among other things — the study of models of dynamic behaviour, the design of incremental compilers and the exploitation of the “growing” language concept.

1 Introduction

How to create systems that exhibit flexibility and dynamic behaviour and cope with changing requirements is an interesting problem. How to create meta-systems (languages, frameworks and so on) that can be used to build those systems is the heart the problem and a very hot topic of research [2,5,6,12,17,18].

It is a mere fact that traditional OOP languages are inadequate to model dynamic behaviour. Inheritance for example is used for a variety of tasks (subtyping, specialization of behaviour, implementation details) and yet these ways are still too fixed (both within a specific language and as general notions) to support dynamically changing requirements. Several attempts have been made in order to remedy the situation, that is enhance the expressiveness of existing languages in order to handle several dimensions for the problem domain decomposition at once.

During the last several years we have been employed in the design and development of software programs either as part of our job responsibilities or out

of personal interest. C++ and Java have been the main implementation languages. These are very popular object-oriented languages with many different characteristics. C++ has multiple inheritance while Java has single inheritance with interfaces. Java has built-in reflection and dynamic loading support. At the very basic level, they are after all Turing machines, but anyone who has programmed in both can recognize that the style a programmer uses to develop programs is different, depending upon which one is chosen. And when it comes to programming dynamic behaviour, ad-hoc solutions have to be invented.

In this paper we try to attack the problem, taking — in our opinion — a more fundamental road than several popular approaches ([6,12] among them). Instead of postulating ad-hoc extensions to OOP we try to identify the very basics of OOP and ask ourselves: what is the *least set* of characteristics we can build upon and what else do we need to make other characteristics *derivable*?

Ultimately, we would like to a) handle emerging design and programming disciplines such as the *Separation of Concerns* principle and b) be as open as possible, so as to cope with *unanticipated evolution*. For this, we choose to build upon OOP, using an object-oriented kernel. After this kernel has been identified, further insight leads to additional results and several research questions.

The rest of the paper is organized as follows: In the following Section we identify areas of work which are close to our research. Proceeding then, in Section 3, we present the proposed framework together with examples that clearly show the motivating ideas. In Section 4 we outline possible directions for research related to our proposed framework. Finally, we conclude in Section 5.

2 Related Work

Our focus on the related work literature is on extensions of the object-oriented model and extensions to programming language concepts. The list is by no means exhaustive.

Adaptive Programming, the Demeter Method is an extension of object-oriented programming “where relationships between functions and data are left flexible, that is, where functions and data are loosely coupled through navigation specifications” [14]. We are currently not addressing the structure-shyness advocated by the Adaptive Programming methods and it is an open question whether and how this might be accomplished.

Aspect-Oriented Programming [12] distinguishes aspects from components which are units of system’s functional decomposition. Aspects are well modularized concerns that crosscut a system’s implementation and functionality. The primary goals of AOP are a) to cleanly separate components and aspects and b) to provide mechanisms for their abstraction and composition in order to build an overall system. The main tool of the AOP philosophy, *AspectJ* [3], relies on syntactic Java extensions and the notion of join-points to provide crosscutting functionality. The syntactic extensions can be modeled with our meta-information notion and their respective run-time actions with appropriate algorithms.

The ATOMA Model [19,18] is an extension to the OO model, having its primary goal as the separation of concerns between the base object behaviour and other, context-dependent behaviour variations. It introduces Java language syntactic extensions based on the concept of role and a respective runtime environment for the execution of code which deals with dynamic (runtime) behavioural evolution of objects.

Composition Filters [5,6] is an AOP technique where an aspect is expressed in filters that intercept and manipulate object communication messages. Behavioural evolution is achieved in a uniform way by the activation of the filters.

Multidimensional Separation of Concerns (MDSOC) [17] builds on previous work on Subject-Oriented Programming [10]. Its main idea is that software concerns “live” in a multidimensional hyperspace [15] and it should be possible to create software by recognizing and combining concerns in any of these dimensions. *Hyper/J* is a tool in this direction and its role is somewhat complementary of *AspectJ*. It works with existing Java code but instead of introducing language extensions it defines the hyperspaces and the concerns relations using external configuration files.

“Growing a Language” Concept. Our proposal for meta-information syntactic additions could benefit from a general trend emerging in language design, advocating that “from now on, a main goal in designing a language should be to plan for growth” [16]. Among the languages which allow such growth are Forth [8,9], T_EX [13], Scheme [11] and Dylan [7]. Forth and T_EX have been pioneering but their low-level characteristics lead to “more work done” in order to support extended language features. The Scheme macro system fits well into the language and is powerful enough to allow arbitrary computations. Dylan is even more flexible, since it permits the languages extensions to be defined by a user-supplied grammar. Recently, the Java Syntactic Extender (JSE) [4] allows a programmer to extend the Java language by a powerful Dylan-inspired macro system. The extensions themselves are programmed in Java.

3 Proposed Framework

3.1 Key Notions

So what are those magic fundamental characteristics? We have chosen for them to be records [1] supporting the usual OO notion of encapsulation and be enhanced with meta-information. The following code snippet is illustrative of record definitions in our approach:

Meta-information may be attached either at the record level or at the attribute/method level. For example we can see the following pieces of meta-information:

- `[[inherits Person]]`,
- `[[overrides Person.getSalary]]` and
- `[[super]]`.

```

record Person /* definition ... */

record Employee [[inherits Person]]
{
  int getSalary() [[overrides Person.getSalary]]
  {
    /* getSalary code */
  }
  ...
}

record Manager [[inherits Employee]]
{
  int getSalary()
  {
    return 2 * [[super]].getSalary();
  }
}

```

Fig. 1. Sample records with meta-information.

With just the above definitions, we do not have any particular behavioural model. The first piece of meta-information seems like an inheritance relationship between records *Employee* and *Person*. The second also seems like a directive related to polymorphism. The third one is a call to the "super class". But these commonly used notions (inheritance, class, "super", polymorphism) are not fixed anywhere in our generic framework. For the code in Figure 1 to work correctly, a compiler that can understand the specified meta-information and a runtime system to execute the produced code is needed. Thus we have arrived at the equation of the title, that is:

$$\textit{Behaviour} = \textit{Records} + \textit{Meta-Information} + \textit{Algorithms}.$$

Records are the basic building blocks of behaviour. They present a well defined encapsulation barrier and are "binders" for the **self** variable, in the traditional OO sense [1].

Meta-Information denote the record/attribute/method inter-relations. In effect, they show how to combine several kinds or aspects of behaviour. Since the notion of meta-information introduces a layer of abstraction, we need concrete models to experiment and build our programs with. For example, as a minor test of the ideas presented here, it was trivial to (re)produce *single inheritance* as a specific model of meta-information and the respective algorithms.

Algorithms use meta-information to actually produce the desired composition of behaviour. We need these algorithms both at compile-time, so as to

understand meta-information and at run-time, in order to execute behaviour¹. Algorithms introduce another layer of abstraction. For example, we could use similar or generic algorithms to deal with similar models of meta-information. "Similar" here could mean just extension in the common sense, as in the case of multiple inheritance vs. single inheritance.

In the following lines we shall try to apply these notions to well-known examples of object-oriented models. Specifically, we will address representation issues for single and multiple inheritance.

3.2 Example OO Models

Single Inheritance OO Model

The code given in Fig. 1 can be used as a reference for class-based single inheritance. The meta-information has been chosen deliberately to "remind" us how we may define classes in a single inheritance based language. Note that the term *classes* is just a matter of interpretation of the model, that is the algorithms used to interpret² the definitions.

Multiple Inheritance

Let us "rephrase" the model of Fig. 1, this time using definitions that resemble familiar multiple inheritance ones.

```
/* record Person and Employee as in Fig. 1*/  
  
record Manager [[inherits Person, Employee]]  
{  
    int getSalary()  
    {  
        return 2 * [[Employee]].getSalary();  
    }  
}
```

Fig. 2. A representation for multiple inheritance.

Class-based multiple inheritance ala C++ is what is represented in Fig. 2. The intention of the overridden method *getSalary()* is to return for a *Manager* a salary twice as much of the salary of a plain *Employee*.

Discussion

As far as definitions are concerned, these two models are almost identical. The main difference then should lie in the algorithms which are responsible for the interpretation of meta-information. The needed algorithms are:

¹ Of course, we additionally have the algorithms declared as part of a record specification.

² The notion of *interpretation* is used with its generic meaning.

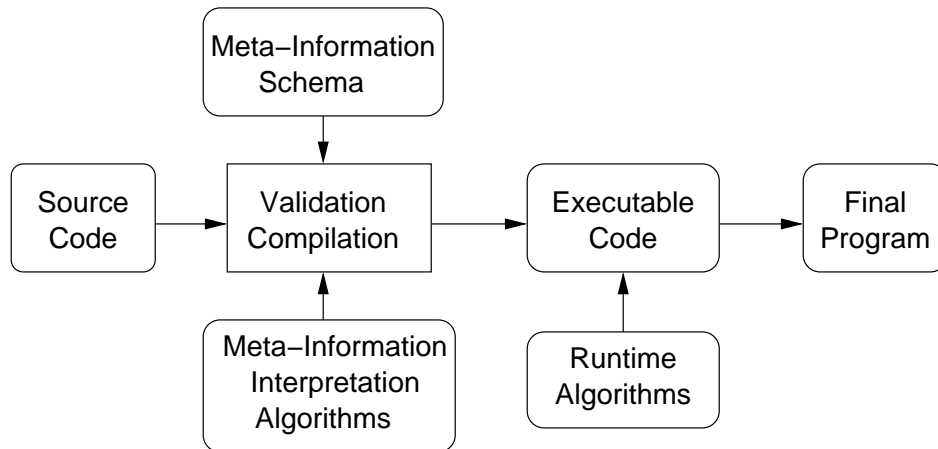


Fig. 3. Framework architecture overview.

1. The one that interprets meta-information and captures the relevant/respective semantics at compile-time, i.e. what is the difference between
 - `[[inherits Person]]` (single inheritance) and
 - `[[inherits Person, Employee]]` (multiple inheritance).
2. The one that actually translates the record definitions into executable code.
3. The supporting algorithms which constitute a runtime system (if this is actually needed for a specific model) for the execution of code. This runtime system could be in the form of libraries (like the standard C++ library), in the form of an elaborate execution environment (like the Java VM) or a combination of the two.

3.3 Architecture Overview

Fig. 3 gives an overview of the proposed framework architectural overview.

A *meta-information schema* (miS) is the set of meta-information needed to specify an object-oriented behavioural model together with their inter-relationships. It should specify at least the following:

1. The “textual” representation, as for example `[[inherits X]]`, where X is a name placeholder.
2. The level that a “textual” representation is applied to, that is: a) record definition, b) method/attribute definition or c) in-method code level.
3. Any relation to some other meta-information schema. For example, a “Multiple Inheritance Schema” (MI-Schema) might reference the “Single Inheritance Schema” (SI-Schema) for reasons such as:
 - The “textual” descriptions of MI-Schema are derived from the ones of SI-Schema.

- The semantics of MI-Schema are based on (are extension of) the semantics of SI-Schema.

The *miS Interpretation Algorithms* are responsible for the validation and compilation of the annotated code (Fig. 1,2). Their output is executable code. But for this code to work, *Runtime Algorithms* are needed to provide a suitable execution environment.

4 Research Agenda

Having chosen the conceptual foundations of our proposal, there is still a great variety of issues that need further clarification. Also, the way to proceed with the proposed framework could have infinitely many paths to follow. An incomplete list of all these issues follows.

Meta-Information Semantics We need a clear definition and semantics of meta-information: In particular, these points need further investigation:

- How meta-information is declared in source code,
- How it is combined with other meta-information, and
- How it is verified for consistency and/or correctness.

Encapsulation The encapsulation barrier of records should be specified exactly. Issues to be addressed are:

- The minimal level of encapsulation built-in in the framework,
- The degree to which this minimal level of encapsulation is affected by user-defined encapsulation models,
- How are extension-models of encapsulation are to be defined and realized in the framework.

Existing Behavioural Models It would be interesting to arrive at already known models of behaviour through our ideas. Single inheritance mentioned previously is clearly just too simple. The main question then, is:

- Can we reproduce the results of Aspect Oriented Programming [12], ATOMA [19,18], Composition Filters [6], Subject-Oriented Programming, Hyperdimensional Separation of Concerns [17]? Can these approaches be defined as special meta-information models accompanied with their respective algorithms?

This particular direction of research may possibly lead to further insight into the expressiveness and effectiveness of our framework and may point to themes for further development.

Model Unification and Clarification If we achieve proper representation and semantics for the several models mentioned, the next obvious step is to fully understand a possible “common denominator”. Speaking optimistically, by breaking the models down to their constituent parts, all the common elements

should emerge and then a scheme that reconstructs the respective models under a unified “description” should be possible.

OOP meets Functional Programming *Records* are representative of object-oriented programming and *Algorithms* hint us toward functional programming. It would be interesting to explore the consequences of combining these two styles. A purely functional approach may even throw (more) light into the functional sub-part of OOP.

Behaviour vs. Structure The approach layed down herein seems behavioural-oriented. How can we handle issues regarding structure? Since we are interested in the dynamic evolution of informational systems, we should address as closely as possible both behavioural and structural evolution. How can structure-shyness [14] be expressed / generalized in our framework? Is it possible? If not, what extensions are needed?

Bootstrapping and Incremental Compilers It would be interesting to immediately arrive at a simple yet extensible meta-information model in order to build a first set of specifications and tools (compilers). Then, we could use the developed system in order to do further research³. This may lead to the development of *incremental compilers*. The use of *incremental* here is not in accordance with the well-known cycle of compile-edit-compile but has to do with the extensibility of the compilers themselves. If this would be achieved then we will have applied the principles of our framework, concerning the evolution of behaviour, to the framework itself!

Dynamic vs. Static Aspects In contrast to other approaches to behaviour composition and evolution, we are primarily interested in starting with its dynamic (runtime) properties and then in specializing to more static ones. In particular, the study of several meta-information models and their respective algorithms may show that particular models handle better than others particular problem families. If we have a dynamic system that can adapt to several problem families then it should be straightforward to optimize the system for specific ones and apply the optimized version to them.

For example, if we express multiple inheritance within our framework and later decide that it works well for the solution of problems in some domain, we should migrate our “schema” of meta-information and algorithms to a fixed language, compiler and runtime system able to handle *only* multiple inheritance. In this “fixed” model the notion of meta-information disappears.

We must note here, that *dynamic* in our framework has a twofold meaning:

1. We do not particularly work with a fixed *Meta-Information Schema* (miS) and
2. The runtime behaviour of an miS + its corresponding algorithms may exhibit dynamic features, as in the case of unanticipated behavioural evolution of a running object/system.

Language Support We need “good” language constructs to facilitate the definition and use of meta-information. Ideally, we need a base language and

³ This is merely the technique of *bootstrapping*.

some core/inherent mechanism that allows us to grow [16] the language. Scheme, Dylan and — lately, with the advent of the Java Syntactic Extender — Java may provide a prototypical development environment. The pros and the cons of each candidate should be carefully measured (i.e. Do we need a strong type system?)

Reflection/Meta Object Protocol Another question is how to support reflection/MOP in the generalized meta-information + algorithms framework. Reflective/MOP systems tend to be “easier” for adaptation and evolution.

Theory The semantic aspects of the meta-information definition, its use and its combination with the algorithmic part of the framework clearly need to be investigated. This issue is related with the *OOP meets Functional Programming* issue discussed above and work from the two fields could be combined.

5 Conclusions

We have proposed a framework whose aim is to study object-oriented behavioural models in a flexible and dynamic way. To our knowledge, this is the first attempt at such a unified way. Although we do not yet have a fully generic implementation, our first prototypical tests indicate that single/multiple inheritance and mixin-based models of behaviour are straightforward to deal with. In our immediate plans is the study of Aspect-Oriented Programming and Multidimensional Separation of Concerns and their respective representative tools *AspectJ* and *Hyper/J*.

We are currently working on a generalized compiler framework that will be able to accommodate generic specifications of meta-information and to facilitate the use of pluggable algorithms for the interpretation and execution of annotated code.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.
2. Adaptive Programming, Demeter Law. <http://www.ccs.neu.edu/research/demeter/>.
3. AspectJ Project. <http://aspectj.org>.
4. J. Bachrach and K. Playford. The Java Syntactic Extender. In *Proc. of the Object-Oriented Programming, Systems, Languages and Applications Conference, OOPSLA*, 2001.
5. L. Bergmans. *The Composition Filters Object Model*. University of Twente, Dept. of Computer Science, 1994.
6. L. Bergmans and M. Aksit. Composing Multiple Concerns Using Composition Filters. *Communications of the ACM*, 44(10):51–57, Oct. 2001.
7. Dylan Reference Manual. http://www.gwydiondylan.org/drm/drm_1.htm, April 1998.
8. FIG (Forth Interest Group). <http://www.forth.org>.
9. Gforth Manual. <http://www.complang.tuwien.ac.at/forth/gforth/Docs-html/>.
10. W. Harrison and H. Ossher. Subject-Oriented Programming - A Critique of Pure Objects. In *Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA93*, pages 441–428, Sept. 1993.

11. R. Kelsey, W. Clinger, and J. Rees. (eds.) Revised⁵ Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), Aug. 1998.
12. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of ECOOP '97*, LNCS 1241. Springer-Verlag, June 1997.
13. D. E. Knuth. *The T_EXbook*. Addison-Wesley, Reading, Massachusetts, 1984.
14. K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method*. PWS Publishing Company, 1996.
15. H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns using Hyperspaces. *IBM Research Report 21452*, Sept. 1999.
16. G. Steele. Growing a Language, Invited Talk. *Proc. of the Object-Oriented Programming, Systems, Languages and Applications Conference, OOPSLA*, 1998.
17. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of Separation: Multidimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. ACM, 1999.
18. D. Theotokis. *Classes and Inheritance: Are they enough for the modeling of dynamically evolving Information Systems?* PhD thesis, Department of Informatics, University of Athens (*in greek*), 2001.
19. D. Theotokis, G. Gyftodimos, and P. Georgiadis. Atoms: A methodology for component object oriented software development in the education context. In Y. Sun, D. Patel, and S. Patel, editors, *International Conference on Object-Oriented Information Systems OOIS96*, pages 226–242. Springer-Verlag, December 1996.