Representing Generalization Relationships in Logic Programming

Pablo R. Fillottrani
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur, Bahía Blanca, Argentina
prf@cs.uns.edu.ar

Abstract

Modeling is one of the most important activities throughout any software development life cycle. Within object-oriented modeling, class generalization is a powerful, yet simple concept for abstracting commonalities between classes. Although, as classes evolved in the model, their semantics may become inconsistent with previous generalization relationships. Automated tools are necessary in order to preserve model coherence. In this paper we present a logic programming translation of several kinds of class hierarchies, in order to be used in the process of automated model checking. This representation includes positive and negative information, and preserves individual class properties. Thus, the resulting program is not affected with changes in class semantics or in the hierarchies.

Keywords: `artificial intelligence, knowledge representation, object-oriented modeling, logic programming applications`

1 Introduction

In this paper we will consider the problem of reasoning about class hierarchies, from the point of view of knowledge representation. A *class* is a specification of the structure and behavior of a collection of values, which are called *objects*. In different contexts this concept can take other names, like *type* in programming languages, or *sort* in several logical systems. A *class hierarchy* is a set of classes related by the *subclass* relationship, such that B being a subclass of A means that B has all the characteristics from A, possibly adding some proper ones. That is to say, the values of B are also values of A. In object-oriented programming this relationship is also denominated *inheritance* or *generalization*. Class hierarchy is a powerful concept, and it has been studied from the point of view of several fields, like artificial intelligence, databases, and object-oriented programming [6, 8, 11]. However, all these frameworks analyze static and complete hierarchies, where values, classes, and relationships do not change and there is no information missing.

In the process of object-oriented modeling, classes are the most important building blocks. Their specification is developed incrementally, starting from informal concepts in the vocabulary of the problem. Properties and relationships are added, modified and remove from classes. Therefore, model checking is necessary to detect inconsistencies with a previous representation. But in this case, we require incomplete and dynamic class semantics. The objective of this work is to present a representation of class hierarchies so that it can be of practical use in such a context.

The paper is structured as follows. First, we introduce the tool that motivates this new representation. Then, we present the taxonomy of hierarchies according to the knowledge status of the values in the classes. Each kind will be translated into a different representation in the following section. Finally, we compare with related formalisms and present some conclusions.

2 Object-Oriented Modeling

Modeling is one of the most crucial activities throughout any software development life cycle. In a general sense, a model is "an abstraction of something for the purpose of understanding it before building it"[12]. These models allow us to reason about the properties that entities possess, and so help us to deal with systems that are too complex to be directly understood.

It is possible to construct different models of the same system, each one emphasizing certain aspects that abstract the system from a particular point of view. Consequently, each model focuses the attention on some details, simplifying others that are not important from this perspective. As a result, it would be easier to understand the whole system as several complementary small models, than as a big complex one.

The same model can be expressed at different levels of abstraction in the process of software development, according to their intended purpose: showing requirements

and domain knowledge, helping in the design of the system, communicating design decisions, organizing complex systems, providing documentation for the final system, etc. Therefore, the choice of the model has a deep influence on how the problem is attacked and how the solution will look like.

In order for such a model to be useful, it is necessary to formulate it in a common language to be shared with all the people involved in the system development. An approach to such a language is the Unified Modeling Language (UML) [1, 13]. UML is a graphical language used to visualize, specify, construct and documentate the elements of a software-intensive system, i.e. for defining models of such a system. By 1995 Booch, Jacobson and Rumbaugh combined their own languages trying to preserve their particular advantages, and created the first rough copies of UML. In January 1997 UML was proposed for standardization to the Object Management Group (OMG). The proposal was modified, and finally approved in November of the same year as version 1.1. After several revisions by the OMG, the version 1.3 was released in 1999.

UML provides notation for expressing the model, in the form of graphic and text elements. Attached to these elements, there is a semantic interpretation that attempts to capture the meaning of the model. Supporting tools for object oriented development in UML should not only provide a graphic editor for the notation, but also help to ensure the coherence of its semantic aspects. The design of the latter is a very difficult issue because the intended meaning depends heavily on the purpose of the model in a specific moment. For example, sometimes may be desirable to check the consistency of the relationships between classes, but later it might be more important to assure the type consistency of every method invocation.

The Software Engineering Laboratory at the Universidad Nacional del Sur is currently working on a tool, called HEMOO (Herramienta para Edición de Modelos Orientados a Objetos)[3], for supporting the development of object oriented models in UML. The objective is to assist the user in the management of the semantic aspects of a model, as well as the notational ones. In this sense the tool supplies flexible semantic checks that can be invoked on demand. Checking on demand means allowing the user to set those defined properties that will be checked, and when this control is done. Flexibility is achieved by providing the user the possibility to define his/her own checks, guaranteeing their treatment on an equal basis as predefined ones. Consequently, it is expected that this automatic control will reduce the manual, error-prone work of maintaining models consistency across all life cycle phases.

3 Problem Description

In this section we consider the positive and negative inferences that can be obtained from the specific characteristics of classes in the hierarchy. Figure 1 shows the class hierarchy that will be analyzed along the whole section. The subclass relationship should always have the property of being a tree, and it can have several levels although in figure 1 we show only two. Although the first property is not checked in the programs,

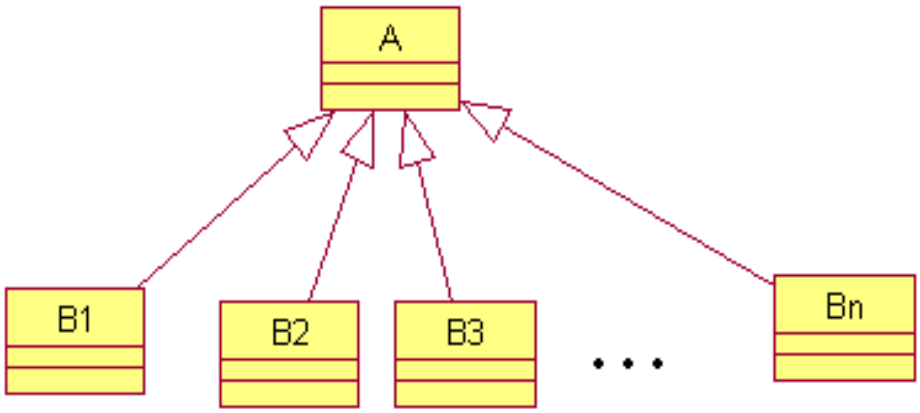

Figure 1: Class hierarchy to analyze.

it can be easily added.

Reasoning about class hierarchies means to deduce properties for the instances of a class starting from the information of classes and the inheritance relationship. In fact, the paradigmatic example of the non-monotonic reasoning, the tweety example [5], is a kind of application of reasoning about class hierarchies. The problem is that not all the hierarchies have the same characteristics, in consequence a reasoning method for one of them is not necessarily adapted for the remaining. The main contribution of this paper precisely consists on introducing a systematic classification for these hierarchies, and to represent them in a logic programming language. This implementations will allow to reason not only about those properties that are satisfied, but also about those that are *not* satisfied. We employ in this case the language of *non-monotonic logic programs*[2, 4] that subsumes both strong negation and negation as failure.

The different kinds of reasoning about hierarchies will be distinguished by two fundamental properties: the quality of knowledge of the instances of each class, and the known relationships among the classes in the hierarchy. To characterize this last point it will be needed the notation $[\![A]\!]$ that will denote the collection of values or instances of A. From the definition of subclass, it is possible to concludes that if B is a subclass of A then $[\![B]\!] \subseteq [\![A]\!]$. Then, combining these two properties we will distinguish different situations that require different reasonings. The examples will follow the hierarchy in the figure 1, but it is immediate to apply the scheme to any other hierarchy. For representing these classes in the language of **nmlp**, the predicates $\mathtt{a}^{(1)}$, $\mathtt{b1}^{(1)}$, $\mathtt{b2}^{(1)}$, ..., $\mathtt{bn}^{(n)}$ will be used. Therefore, we presuppose they belong to the signature in every **nmlp** in this section.

We will introduce the properties of the following classes of class hierarchy: *partition*, *taxonomy*, and *views*. The implementation will emphasize the distinctive features of each category, and motivated this classification. We will eventually have some variants

according to the quality of knowledge for the class or the subclasses.

4 Representation of Class Hierarchies

The first situation to be analyzed is called *partition*, and it is characterized by the property that $\bigcup_i [\![B_i]\!] = [\![A]\!]$ and for all $i \neq j$, $[\![B_i]\!] \cap [\![B_j]\!] = \emptyset$. This is, the subclasses completely define all the values of the class, and common values don't exist among the subclasses. Mathematically, a partition is the quotient set generated by an equivalence relation. In this case the equivalence relation is determined by membership to the same subclass.

Depending on the availability or not of exact knowledge of the instances of each subclass, two different programs are presented.

Definition 4.1 We call the *partition definition with exact knowledge* to the following rules

$$
\begin{array}{lll}
1. & \mathtt{a}(X) & \leftarrow \ \mathtt{b1}(X) \\
2. & \mathtt{a}(X) & \leftarrow \ \mathtt{b2}(X) \\
 & \ldots & \\
n. & \mathtt{a}(X) & \leftarrow \ \mathtt{bn}(X) \\
n+1. & \neg\mathtt{a}(X) & \leftarrow \ \neg\mathtt{b1}(X), \neg\mathtt{b2}(X), \ldots, \neg\mathtt{bn}(X) \\
n+2. & \mathtt{min}(\mathtt{b1}(X)) & \leftarrow \\
n+3. & \mathtt{min}(\mathtt{b2}(X)) & \leftarrow \\
 & \ldots & \\
2n+1. & \mathtt{min}(\mathtt{bn}(X)) & \leftarrow
\end{array}
$$

for every class A with subclass B_i, together with rules

$$\neg\mathtt{bi}(X) \ \leftarrow \ \mathtt{bj}(X)$$

for every pair of subclasses $B_i \neq B_j$.

The first n rules correspond to the definition of subclasses, the $n+1$th. rule defines the completeness of the subclasses in covering the class, and rules $n+2$–$2n+1$ states the exact knowledge of each subclass values. Finally, there are rules to represent the exclusivity of each subclass. These last rules are not necessary to infer negation information, since non monotonic inference is enough, but rather they are used to generate inconsistency in the event of data errors.

In order to formalize a partition with incomplete knowledge of instances, it is enough to remove from the previous definition the non-monotonic inference that makes complete each B_i extension.

Definition 4.2 We call the *partition definition with incomplete knowledge of* B_i to

the **nmlp** that contains the following rules

$$\begin{array}{lll} 1. & \mathtt{a}(X) & \leftarrow \ \mathtt{b1}(X) \\ 2. & \mathtt{a}(X) & \leftarrow \ \mathtt{b2}(X) \\ & \ldots & \\ n. & \mathtt{a}(X) & \leftarrow \ \mathtt{bn}(X) \\ n+1. & \neg\mathtt{a}(X) & \leftarrow \ \neg\mathtt{b1}(X), \neg\mathtt{b2}(X), \ldots, \neg\mathtt{bn}(X) \end{array}$$

for each class A with subclasses B_i, together with rules

$$\neg\mathtt{bi}(X) \ \leftarrow \ \mathtt{bj}(X)$$

for each pair of subclasses $B_i \neq B_j$.

In this circumstance, it will be possible to have the consequence $\mathtt{a}(t)$ without the corresponding $\mathtt{bi}(t)$ for every i. In order to be so, however, it is necessary that no $\neg\mathtt{bi}(t)$ belongs to the consequences.

Next we show an example of this class hierarchy.

Example 4.3 Let us suppose that we have a series of bank account transactions such as deposits, withdrawals, transfers, each with its own details in addition to the more general characteristics of every transaction. Then the partition definition with exact knowledge with the following facts constitute the **nmlp** Π.

$$\begin{array}{ll} 1. \ \mathtt{transferencia(op2234)} & \leftarrow \\ 2. \ \mathtt{depósito(op0936)} & \leftarrow \end{array}$$

Program Π has the expected consequences

$$\begin{array}{l} \{ \ \mathtt{operación(op2234)}, \mathtt{operación(op0936)}, \\ \quad \neg\mathtt{depósito(op2234)}, \neg\mathtt{transferencia(op0936)}), \\ \quad \neg\mathtt{extracción(op2234)}, \neg\mathtt{extracción(op0936)} \ \} \end{array}$$

Notice the flexibility of this program is provided by the possibility of adding new individual transactions without changing the existing program, as well as the possibility of adding new classes of transactions that only require the natural adaptions to the definition. ∎

The following kind of hierarchies to be represented is called *taxonomy*. In a taxonomy, likewise a partition, the subclasses don't have common instances, that is to say $[\![B_i]\!] \cap [\![B_j]\!] = \emptyset$ if $i \neq j$. But it differs from partition in that a precise classification of all the instances doesn't always exist. Therefore, some subclasses might exist without being explicitly named in the hierarchy. For example, exists $x \in [\![A]\!]$ but $x \notin [\![B_i]\!]$ for every $i, 1 \leq i \leq n$; thus x belongs to an unnamed subclass. For being partition it is necessary that the subclasses are completely specified.

An internal classification for taxonomies is also possible considering the level of knowledge of each subclass. But in this case, since the subclasses are not complete,

the uncertainty can be given at level of the subclasses, or at level of the most general class. Then it will be defined taxonomies by all the possibilities from combining exact knowledge and incomplete knowledge of the subclasses and of the general class. We start the definitions from the most incomplete variant, in way of adding rules corresponding to the new information.

Definition 4.4 We call the *taxonomy definition with incomplete knowledge* to the **nmlp** that contains the rules

$$\begin{array}{lll} 1. & \mathtt{a}(X) & \leftarrow \quad \mathtt{b1}(X) \\ 2. & \mathtt{a}(X) & \leftarrow \quad \mathtt{b2}(X) \\ & \ldots & \\ n. & \mathtt{a}(X) & \leftarrow \quad \mathtt{bn}(X) \end{array}$$

fore every class A with subclasses B_i, and the rules

$$\neg\mathtt{bi}(X) \quad \leftarrow \quad \mathtt{bj}(X)$$

for every pair of subclasses $B_i \neq B_j$.

Essentially, this program only allows monotonic positive inferences, except for the negative information generated by subclass exclusivity. The program assigns the truth value **indefinido** to most literals in $\mathtt{a}(t)$ and $\mathtt{bi}(t)$ when there isn't enough information about t.

Now we present the versions that have incomplete knowledge at one level in the hierarchy, and complete knowledge in the other.

Definition 4.5 We call the *taxonomy definition with incomplete knowledge of* B_i to the **nmlp** that adds to the taxonomy definition with incomplete knowledge the following rule

$$1. \quad \mathtt{min}(\mathtt{a}(X)) \quad \leftarrow \quad \neg\mathtt{b1}(X), \neg\mathtt{b2}(X), \ldots, \neg\mathtt{bn}(X)$$

for class A with subclasses B_i.

This new rule is used to infer negative literals such as $\neg\mathtt{a}(t)$ for every t that it is known it doesn't belong to any B_i. This fact doesn't exclude the existence of instances of A that don't have a known subclass in the B_i's.

We applied the same technique in the case of exact information about the subclasses instances, but inexact about A's instances. The implementation adds a non-monotonic inferences for each subclass.

Definition 4.6 We call *taxonomy definition with incomplete knowledge of* A to the **nmlp** that merges the taxonomy definition with incomplete knowledge with the rules

$$\begin{array}{ll} 1. & \mathtt{min}(\mathtt{b1}(X)) \quad \leftarrow \\ 2. & \mathtt{min}(\mathtt{b2}(X)) \quad \leftarrow \\ & \ldots \\ n. & \mathtt{min}(\mathtt{bn}(X)) \quad \leftarrow \end{array}$$

for each subclass B_i.

Finally, a taxonomy with complete knowledge of its values is the class of hierarchy that allows the inference of more negative literals. We give this implementation adding minimization for the class A as well as the subclasses B_i.

Definition 4.7 We call the *taxonomy definition with complete knowledge* to the **nmlp** that adds to the taxonomy definition with incomplete knowledge for A the following rule

$$1.\quad \texttt{min}(\texttt{a}(X)) \quad \leftarrow$$

This circumscriptive policy is applied to every term t, meaning that if t is an A then we know the fact, and if it isn't an A we also know the fact. On the other hand, if we consider a taxonomy with incomplete knowledge of the B_i's the minimization is only applied in the case of an explicit negative knowledge for each subclass.

Example 4.8 We want to represent a classification of modern European languages in a program. A sample set of facts can be

$$\begin{array}{lll}
1. & \texttt{romance(spanish)} & \leftarrow \\
2. & \texttt{romance(french)} & \leftarrow \\
3. & \texttt{romance(italian)} & \leftarrow \\
4. & \texttt{germanic(english)} & \leftarrow \\
5. & \texttt{germanic(german)} & \leftarrow \\
6. & \texttt{slavonic(russian)} & \leftarrow \\
7. & \texttt{slavonic(polish)} & \leftarrow \\
8. & \texttt{celt(irish)} & \leftarrow \\
9. & \texttt{celt(gaelic)} & \leftarrow \\
10. & \texttt{language(basque)} & \leftarrow \\
11. & \texttt{language(greek)} & \leftarrow
\end{array}$$

Let us suppose all these languages are provided as facts in the predicate $\texttt{language}^{(1)}$, and some of them are classified as `romance`, `germanic`, `slavonic` and `celt`. Some languages have not been classified in any group. Then we have a taxonomy with incomplete knowledge of $\texttt{language}^{(1)}$.

These facts, together with the taxonomy definition with incomplete knowledge of $\texttt{language}^{(1)}$, produce the following expected consequences

$$\{\ \texttt{language(english)}, \neg\texttt{celt(french)}, \neg\texttt{slavonic(greek)}, \neg\texttt{romance(greek)}\ \}$$

Notice that actually in the cases of `greek` and `basque` we can prove the negation of every subclass, even though this does not contradict the belonging to the general class.

It is clear that this scheme can be used to add o remove new languages, or even new classes of languages. The resulting program will have as conclusions all correct positive and negative inferences. ∎

The last class hierarchy that will be analyzed is denominated *view*. A subclass is a view of a more general class if it can share values with one or more other subclasses

of the same general class. In other words, it may be that $[\![B_i]\!] \cap [\![B_j]\!] \neq \emptyset$ for $i \neq j$. The intuitive meaning is that the same instance can be "seen" as belonging to several subclasses, or views, at the same time. In general views correspond to subclasses with different level of abstraction, which makes it necessary a combination of all the subclasses in the same level with all the subclasses of another level (see example 4.11).

We can distinguish in views two situations: in the first one, membership of an element to a subclass doesn't generate any supposition about its ownership to another subclass; in the second, membership of a subclass supposes by default that the value does not belong to any other subclass. The first is called *simple view* and the second *strict view*.

Definition 4.9 We call the *simple views definition* to the following **nmlp**:

$$\begin{array}{lll} 1. & \mathtt{a}(X) & \leftarrow \ \mathtt{b1}(X) \\ 2. & \mathtt{a}(X) & \leftarrow \ \mathtt{b2}(X) \\ & \ldots & \\ n. & \mathtt{a}(X) & \leftarrow \ \mathtt{bn}(X) \\ n+1. & \mathtt{min}(\mathtt{a}(X)) & \leftarrow \end{array}$$

for each class A with subclasses B_i.

The definition of simple views supposes that knowledge of the instances of the class is complete, and therefore non-monotonic inference is applied to the class predicate. The definition of strict views, in addition, needs non-monotonic inference for the subclasses predicates for all subclasses, when the term is already known that it belongs to a different subclass.

Definition 4.10 We call the *strict views definition* to the **nmlp** defined by adding to the simple views definition of A with subclasses B_i the rules

$$\mathtt{min}(\mathtt{bi}(X)) \ \leftarrow \ \mathtt{bj}(X)$$

for each pair of different subclasses $B_i \neq B_j$.

Example 4.11 Let us suppose that we have a class represented by $\mathtt{employee}^{(1)}$, with the subclasses represented by: $\mathtt{temporary}^{(1)}$, $\mathtt{permanent}^{(1)}$, $\mathtt{technician}^{(1)}$ and $\mathtt{administrative}^{(1)}$. The first pair of subclasses correspond to a classification according to the contract that ties the employee with the company, while the second pair classify the class of work they do. Evidently, both classifications are orthogonal, since we can have temporary administrative, permanent administrative, temporary technician and permanent technician employees. It is reasonable to suppose that given an employee, it is known both characteristics. So the problem suits to the definition of strict views, and the programs adds the following specific facts:

$$\begin{array}{ll} 1. & \mathtt{temporary(ernest)} \ \leftarrow \\ 2. & \mathtt{technician(ernest)} \ \leftarrow \\ 3. & \mathtt{administrative(ernest)} \ \leftarrow \\ 4. & \mathtt{permanent(susan)} \ \leftarrow \\ 5. & \mathtt{administrative(susan)} \ \leftarrow \end{array}$$

Notice that even the classification of instances with the same criterion is no exclusive, since the first employee is both technician and administrative. Then, consequences from this **nmlp** are, besides the facts from the program, the following literals

$$\{\ \neg\texttt{temporary}(\texttt{susan}), \texttt{technician}(\texttt{susan}),\ \neg\texttt{permanent}(\texttt{ernest}), \texttt{employee}(\texttt{susan}),\ \texttt{employee}(\texttt{ernest})\ \}$$

The addition of a new view, even though without any instances, is completely consistent with this program. ∎

So far, we have presented some common class hierarchies, but the presentation doesn't pretend to be exhaustive. Surely some applications require hierarchies with other characteristics. One of the main advantages of this analysis, and the implementations, is that some common properties can be isolated, like disjoint subclasses, complete subclasses, complete or incomplete knowledge, etc. Later on, these common properties can be combined to formalize a new hierarchy that matches the specific requirements. Moreover, it is possible to present a hierarchy by merging the proposed classification. For example, in the last example we can first define a taxonomy for each criteria, and the merge the two taxonomies with a view definition. By identifying and documenting the uses of each class, a programmer in this language would have available a diverse library with all these implementations.

5 Related Work and Conclusions

We presented a general description of a logic programming representation for class hierarchies, in order to be used in an automatic tool for object-oriented modeling. This representation emphasizes semantic controls on the model under consideration. The implementation of the tool takes advantage of the declarative nature of logic programming paradigm, and the well-known systems that implement its semantic.

As pointed out in the introduction, formalizations for class hierarchies have been realized in several contexts. However, object-oriented modeling requires incomplete and dynamic knowledge representation, which is not considered in other approaches. In Artificial Intelligence, logic programming formalizations for inheritance nets include *contextual logic programs* [9, 10] that addresses inheritance of logic programs; and L&O [7] that translates hierarchies into traditional logic programs. The main problem of these approaches is that they do not allow to reason about the properties that *aren't* present in a class, nor they allow incomplete knowledge. Also, we do not know of proof procedures for these languages. Several other ad-hoc logics [14, 6] are so expressive that are intractable. In object-oriented programming languages class representation do not have formal semantics, so it is not possible to represent formal properties and their combination.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide*. Addison-Wesley, Reading, MA, 1999.

[2] P. R. Fillottrani. Sobre la negación y la inferencia no monótona en la programación en lógica. In *Proceedings CACIC'00, Sexto Congreso Argentino de Ciencias de la Computación*, pages 489–500, 2000.

[3] P. R. Fillottrani, E. C. Estévez, and S. Kahnert. Applying logic programming techniques in a tool for object-oriented modeling in UML. In *Proceedings of the 13th. International Conference on Software Engineering and Knowledge Engineering*, pages 228–335. Knowledge Systems Institute, 2001.

[4] P. R. Fillottrani and G. R. Simari. Circumscriptive logic programming. In *Proceedings of the XIV International Conference of the Chilean Computer Science Society*, 1994.

[5] D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors. *Handbook of logic in artificial intelligence and logic programming*, volume 5. Oxford University Press, 1998.

[6] J. F. Horty. Some direct theories of nonmonotonic inheritance. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of logic in artificial intelligence and logic programming*, volume 3, pages 111–187. Oxford University Press, 1994.

[7] F. G. McCabe. An introduction to *L&O*. In K. R. Apt, J. W. de Bakker, and J. J. Rutten, editors, *Logic programming languages: constraints, functions and objects*, pages 148–184. MIT Press, 1993.

[8] B. Meyer. *Object oriented software construction*. Prentice Hall, 2 edition, 1997.

[9] L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Proceedings of the 6th. International Logic Programming Conference*, pages 284–299. MIT Press, 1989.

[10] L. Monteiro and A. Porto. A language for contextual logic programming. In K. R. Apt, J. W. de Bakker, and J. J. Rutten, editors, *Logic programming languages: constraints, functions and objects*, pages 115–147. MIT Press, 1993.

[11] F. Pfenning, editor. *Types in logic programming*. MIT Press, 1992.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design*. Prentice Hall, 1991.

[13] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, 1999.

[14] D. Touretzky. *The mathematics of inheritance systems*. Morgan Kaufmann Publishers, 1986.