MODULARITY: U: Relations in Role-Based Data Modeling, Navigation and Updates

Daco C. Harkes
Delft University of Technology
d.c.harkes@tudelft.nl

Abstract

Object-oriented programming languages support concise navigation of relations represented by references. However, relations are not first-class citizens and bidirectional navigation is not supported. The relational paradigm provides first-class relations, but with bidirectional navigation through verbose queries. We present a systematic analysis of approaches to modeling and navigating relations. By unifying and generalizing the features of these approaches, we developed the design of a data modeling language that features first-class relations, n-ary relations, native multiplicities, bidirectional relations and concise navigation. This paper is a shortened version of work published in [9]. We extend our previous work by ongoing work on multiplicity-safe update operations.

1. Introduction

Object-Oriented programming languages model data with object graphs. Navigation through object graphs is simple; following references leads to related objects. But references in object graphs are one-directional and cannot be navigated backwards. Bidirectional navigation can be obtained by storing references on both sides of relations between objects. But keeping such redundant references consistent requires bookkeeping code. By contrast, relational databases support bidirectional navigation. Foreign keys can be used in queries to navigate both ways. There is no need for redundant references. Queries are however not as concise as navigation through references.

Proposals for Object-Oriented languages with first-class relations provide bidirectional navigation [3]. These languages remove the need for manually keeping references consistent but navigation is done through querying, which is still verbose. There are modeling techniques that are yet different from Object-Oriented and relational modeling: Object-Role modeling [7], Entity-Relationship modeling [5], UML [12] and undirected graphs.

In this paper, we present a systematic analysis of the design space of relations in data modeling and present a new data modeling language that unifies and generalizes relations. In particular, our contributions are:

- We extrapolate Steimann's approach [20] to model multiplicities using annotations in Java to *native multiplicities* that are integrated into the type system (Section 2).
- A systematic analysis of approaches to modeling relations (Section 3).
- A new relational data modeling language featuring native multiplicities, bidirectional navigation, n-ary relations, first-class relations, and concise navigation expressions based on the analysis (Section 4).

```
class Student { }

class Course {
  @any(ArrayList.class) Student student;

  void addStud(@any(ArrayList.class) Student s){
    this.student += s;
  }
}
```

Figure 1. Multiplicity annotations in Java

- An overview of approaches for multiplicity-safe data updates (Section 5).

A formal definition of the type system and operational semantics of this language is available in the extended version [9].

2. Native Multiplicities

The first thing we need to fix to get relations right is the treatment of their cardinality or *multiplicity*. Encoding of *to-many* relations as associations to collections results in a discontinuity in programming style [20]:

- Navigating *one-to-one* and *many-to-one* relations produces singleton values, while navigating through *one-to-many* and *many-to-many* relations produces collections of values. Thus, the caller has to unwrap the result before using it, for example by using an iterator.
- The caller has to deal with different sub-type substitution conditions. Suppose `Student` extends `Person`. Assigning an `Student` to a `Person` is fine (*to-one*), but trying to assign `Set<Student>` to `Set<Person>` will trigger a type error (*to-many*).
- The call semantics is call-by-value for *to-one* and call-by-reference for *to-many*. Collection objects are passed by reference, so that they can be modified by the callee. Call-by-value semantics for collections requires immutable collections.

Multiplicity Annotations To address these issues, Steimann proposes an extension of regular Object-Oriented programming with multiplicities [20]. He presents an extension of Java with multiplicity. Expressions of a singleton value type can return an arbitrary number of objects of this type. Figure 1 illustrates the approach with a small example in which a `Course` has an association to `Student`. Through the `@any` annotation the association is declared to be to-many instead of using a collection type.

```
class Student {
  String! name;
  Course* courses;
  int! numCourses(){ return count(this.courses); }
}
class Course {
  Student* students;
  void addStudent(Student+ s){
    this.students += s;
  }
  int? avgNumCourses(){
    return avg(this.students.numCourses());
  }
}
```

Figure 2. Native multiplicities in Java

Native Multiplicities We have extrapolated Steimann's annotations based approach and integrated multiplicities into the type system to arrive at *native* multiplicities. Type expressions use one of the following four multiplicity operators (similar to regular expressions) to denote the possible range of values:

- `t?` is $[0, 1]$ an optional value of type `t`
- `t!` is $[1, 1]$ a required value of type `t`
- `t*` is $[0, n)$ zero or more values of type `t`
- `t+` is $[1, n)$ one or more values of type `t`

The `!` can be omitted as $[1, 1]$ is the default multiplicity.

As a sketch, Figure 2 illustrates native multiplicities in an extension of Java. We have not formalized an extension of Java, but rather integrated native multiplicities in our relational data modeling language. We formalize a type system for that language including multiplicities. The type system ensures that the actual number of values at run-time is always inside the specified range. For example, assigning an optional string (a value of type `String?`) to a `student.name` will trigger a type error: *multiplicity error:* $[1, 1]$ *expected,* $[0, 1]$ *given*. Our language also supports expected multiplicities for function arguments. The built-in function `count` handles any multiplicity and any type and it returns exactly one integer with the number of values passed. The built-in function `avg` also handles $[0, n)$ values and the argument type must be numeric. The return multiplicity of `avg` depends on its input multiplicity. If a programmer supplies $[0, n)$ as input the return multiplicity will be $[0, 1]$. The average of no values does not exist, so no value will be returned in that case. If the programmer supplies $[1, n)$ as input the return multiplicity is $[1, 1]$. With at least one value there is always an average computable. We use this model of multiplicities, reasoning over ranges, in the type system of our language.

3. Design Space for Role-Based Relations

There are several proposals in the literature for extending data modeling to better support data modeling with relations. This section presents a systematic analysis of the design space of relations in data modeling taking in into account these proposals. Figure 3 summarizes the complete design space in tabular form emphasizing its regularities. The design space consist of two dimensions: the modeling paradigm (columns) and the relation models (rows). For both dimensions we describe the features supported for relations. From this analysis a new data modeling language emerges which unifies and generalizes the various approaches to modeling relations.

The running example data model defines `Students` who are enrolled in `Courses`, sometimes via a first-class `Enrollment` relation. In expressions we use `Student` 'bob' and `Course` 'math'.

	OO	Rel.	ORM	Graph	Fc.	N-ary
Edge			WebDSL			
Tuple		RelJ			✓	
Object		Rumer	our work		✓	✓
Concise	✓		✓			
Bidir.		✓	✓	✓		

Figure 3. Design Space with on the side and bottom the supported features: First-class citizenship (fc), N-ary relations, Concise navigation and bidirectional navigation (bidir).

3.1 Columns: Four Modeling Paradigms

The four columns in the design space represent four modeling paradigms.

Object-Oriented Relations between objects are defined through reference valued attributes, which can be navigated in one direction only. The name of the relation is the name of the attribute in the source class. The relation is unknown to the target class. A relation can also be modeled by, redundantly, maintaining a reference attribute on the other side of the relation, as well, allowing bidirectional navigation. However, this requires code for keeping the two sides of the relation consistent. We do not cover models with redundant information in our design-space analysis, as this is an undesirable property.

Relational In a relational database schema references are expressed as foreign keys; an identifier corresponds to a memory address and a foreign key to a reference into memory. An important difference is that these references can be navigated in two directions through queries in a query language (SQL). ER and UML diagrams are also located in this column, but they only provide schema definitions, not queries. Because queries are verbose we introduce our own notation for forward and backward navigation through references. For forward navigation we use the the normal field access notation. For backward navigation from an object `o` we need to find all the objects of type `T` that refer to `o` through references `r`, which is expressed by `o<-(T.r)`. For example, to find the students enrolled in a course `math` we use the navigation expression `math<-(Student.courses)`.

Object-Role Modeling A distinguishing feature of ORM [7] is that associations between objects have a different name on both sides. This conceptually solves the problem of not being able to refer to a reference backwards. Similarly, inverse properties in WebDSL [21] tie two fields in different classes together as inverses.

Graph databases In contrast to the directed edges in the previous three paradigms, graph databases feature undirected edges. In this model the edge names are defined in both source and target namespaces. As with the ORM paradigm there is always a name available in the namespace of participating objects, but in this case this name is identical for both sides. There is one disadvantage of this model: modeling asymmetric same type relations is nontrivial. Consider a `TreeNode` with a parent and children. If a node `p` has a parent edge to another node `q`, then `q` also has a parent edge to `p`. This can be solved through indirection, but that is not particularly elegant. So we do not consider undirected graphs further on.

3.2 Rows: Three Relation Models

The three rows in the design space correspond to three ways of modeling a relation.

Edge The simplest way of representing a relation is through an edge between two nodes (either directed or undirected). This is a concise way of specifying a relation but it has the disadvantage that the relation is not a first-class citizen (see below). Also it is not possible to declare ternary, or higher arity, relations with edges.

Tuple (Ordered Roles) By lifting relations to objects they become *first-class citizens*, i.e. relations can have attributes, and relations can be the subject in other relations. A relation object modeled as a tuple has ordered roles. The absence of role names requires the order (or position) of the roles to be used for navigation. For binary relations this entails four predefined navigation operators. But for higher arity relations 2^n operators are required, which does not scale.

Object (Named Roles) Giving the roles in a relation names makes navigation understandable and makes modeling n-ary relations feasible.

3.3 Detailed Description of Points in Design Space

We discuss some of the points of the design space.

Relational Tuples: RelJ The RelJ Java extension lifts relations to tuple objects [4]. In RelJ different operators are used to disambiguate between different navigation operations (Figure 4). RelJ provides no facilities for bidirectional navigation. However, that is not a conceptual limitation. Adding two operators (`:.` and `::`) would allow backward navigation. While this is theoretically extensible to relations with more than two participants, it requires adding new operators for each participant.

Relational Objects: Rumer and RelJ extension Naming roles allows usable extension to n-ary relations. This is the model used by Rumer [2, 3] as illustrated in Figure 5. While Rumer's implementation does not support n-ary relations, it provides the ingredients needed for n-ary relations: role names and first-class citizenship. A proposed extension for RelJ [23] adds names to roles, as illustrated in Figure 6, and is essentially equivalent to Rumer's syntax. As an alternative query syntax, we propose `math<-(Enrollment.course).student`, which is closer to the usual navigation syntax: from an object (`math`) find all relations with that object in one of its roles (`Enrollment.course`), and produce objects in the other role (`student`). All these notations are rather verbose, even if more concise than full blown SQL queries. We would prefer a more concise notation for navigating n-ary relations.

ORM Edges: Inverse Properties WebDSL [21] supports bidirectional navigation without a verbose syntax for inverse lookups by means of *inverse properties* [10] as illustrated in Figure 7. Explicit names on both sides of an association simplifies navigation to just following named references. These names have to be defined in both the source and target class.

ORM Objects: this paper Combining the advantages of representing the relation as object and Object-Role Modeling for naming roles, we arrive at our proposal for a unified and generalized approach to modeling relations (Figure 8). Relations are first-class citizens: (1) relations can have attributes and (2) relations can be the subject in other relations. In addition, relations can have any number of roles (n-ary relations). By explicitly providing a name for the navigation between each pair of participants in the relation we get concise navigation expressions: (1) from relation to participant and back (`b_takes_m.student` and `bob.enrollments`), and (2) from participant to other participant (`bob.courses`) and back (`math.students`). Instead of defining these names in the source and target classes, as in WebDSL, all names are introduced in the relation. The declaration of a role `T r <- m i` introduces a role `r` of type `T` with inverse `i` with multiplicity `m`. This provides navigation from relation to participant through `r` and navigation from participant to relation through `i`. A declaration `r1.n1 <-> r2.n2` introduces names for navigation between participants: `r1.n1` leads to `r2` and `r2.n2` leads to `r1`. In contrast to WebDSL,

```
class Student { }
class Course { }
relationship Enrollment (Student, Course) {
  int grade;
}
bob.Enrollment       // bob's courses
bob:Enrollment       // Enrollment-type relations
bob:Enrollment.grade
b_takes_m.from       // bob
b_takes_m.to         // math
```

Figure 4. First-class citizen tuple based relations in RelJ [4]

```
class Student { }
class Course { }
relationship Enrollment
  participants (Student student, Course course) {
  int grade;
}
Enrollment.select(s_c: s_c.course==math).student;
```

Figure 5. First-class relations with named roles in Rumer [2, 3]

```
class Student { }
class Course { }
relationship Enrollment extends Relation
  (Student student, Course course, Student tutor){
  int grade;
}
Enrollment[course == math].student; // math studs
```

Figure 6. Ternary relation extension proposal for RelJ [23]

```
entity Student { courses : Set<Course> }
entity Course {
  students :Set<Student> (inverse=Student.courses)
}
math.students  // math students
bob.courses    // bob's courses
```

Figure 7. Inverse properties in WebDSL [21]

```
entity Student { }
entity Course { }
relation Enrollment {
  Student student <- * enrollments
  Course  course  <- + enrollments
  student.courses <-> course.students
  Int grade
}
bob.courses       // bob's courses
bob.enrollments   // Enrollment-type relations
b_takes_m.student // bob
```

Figure 8. Relations with concise navigation (this paper)

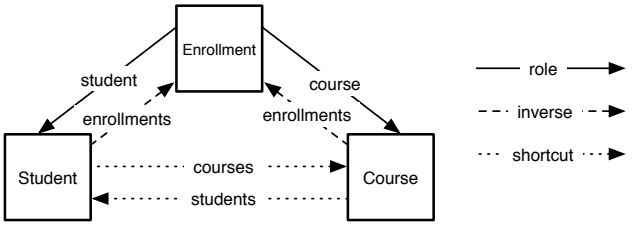

Figure 9. Schematic notation of Figure 8

```
relation Enrollment { Student* Course+ }
```

expands to (lower case participant type, lower case relation type, add s for * and +)

```
relation Enrollment {
  Student student <- * enrollments
  Course  course  <- + enrollments
}
```

expands to (use role name, add s for * and +)

```
relation Enrollment {
  Student student <- * enrollments
  Course  course  <- + enrollments
  student.courses <-> course.students
}
```

Figure 10. Expansion of concise relation definition

```
entity Student {
  Int? avgGrade = avg( this.enrollments.grade )
}
```

Figure 11. Relations language with derivation

these declarations do not introduce attributes in the participant classes, but rather inverses and shortcuts (Figure 9). For example, `bob.courses` is a shortcut for `bob.enrollments.course`.

4. A Relational Data Modeling Language

We have designed a language for data modeling featuring native multiplicities, bidirectional navigation, n-ary relations, first-class relations, and concise navigation expressions based on 'our work' in the design space. In this section we discuss two extensions of the basic idea and the grammar of the language.

Concise Definition of Relations While navigation in Figure 8 is very concise, the definition of a relation is somewhat verbose due to the introduction of names for each of the possible navigation steps. In many cases we can derive these names from the types of the roles. Figure 10 illustrates how a definition with implicit names is expanded to a definition with explicit names. This automatic expansion can of course lead to name collisions, for example if the participant classes have an attribute with a name introduced by a relation. In this case the programmer has to (partially) specify names explicitly.

Derived Attributes To express business logic in data models, we extend entities and relations with *derived attributes*. The value of a derived attribute is described in terms of the values of other attributes and relations as illustrated in Figure 11. Thus, if one of the underlying values changes, the derived attribute is updated.

Grammar The grammar of the relations language is given in Figure 12. a, i, r and t are respectively attribute, inverse, role and entity-type names. The roles, r, are the solid arrows in the design space diagram and the inverses/shortcuts, i, are the dashed and dotted arrows. a', i', r', r'', and t' refer to these names. The lookup expression (`t [ a == e ]`) is only intended to look up objects of a certain type with a certain attribute value in the heap. It is not our intention to provide a full-fledged query language; our focus is on navigation expressions.

Prototype We have implemented this language on the language workbench Spoofax [13]. The prototype is publicly available.[1]

[1] https://github.com/metaborg/relations tag v0.2.5

$$
\begin{aligned}
Program &::= \text{model } Entity^* \text{ execute } e \\
Entity &::= \text{entity } t \text{ \{ } Attribute^* \text{ \}} \\
&\quad | \text{ relation } t \text{ \{ } Attribute^* \ Role^* \ Shortcut^* \text{ \}} \\
Attribute &::= p \ m \ a \\
&\quad | \ p \ m \ a = e \\
Role &::= t' \ r <- m \ i \\
Shortcut &::= r' \ . \ i \ <-> \ r'' \ . \ i \\
p \in PrimitiveType &::= \text{Boolean} \mid \text{Int} \mid \text{String} \\
m \in Multiplicity &::= ? \mid ! \mid * \mid + \\
e \in Expr &::= f \ (\ e \ ) \ \mid \ e_1 \oplus e_2 \ \mid \ ! \, e \ \mid \ e_1 \, ? \, e_2 : e_3 \\
&\quad | \ e \, . \, a' \ \mid \ e \, . \, i' \ \mid \ e \, . \, r' \\
&\quad | \ \text{true} \mid \text{false} \mid literalInt \mid literalString \\
&\quad | \ \text{this} \mid t \, [\, a == e \,] \\
f \in AggrOp &::= \text{min} \mid \text{max} \mid \text{avg} \mid \text{sum} \mid \text{concat} \mid \text{count} \mid \text{conj} \mid \text{disj} \\
\oplus \in \{ &+, -, *, /, \%, \&\&, ||, >, >=, <, <=, ==, ! =, <+, ++ \}
\end{aligned}
$$

Figure 12. The grammar of the relations language

5. Multiplicity-Safe Updates

With data model and navigation in place we can look at a language mechanism for updating data. An update mechanism should preserve the multiplicity invariants specified in the model; it should be *multiplicity-safe*. In this section we will cover existing approaches for multiplicity-safe updates and the requirements for updates in our language.

Multiplicity-Safe Statements Java with multiplicity annotations [20] features the Object-Oriented way of updating state: assignment statements. This limits object graphs that can be constructed while maintaining multiplicity invariants. In our example `Course` has + enrollment relations. The only way to create a new course is to pass a student in the constructor, creating a course without a student would violate the invariant. If `Student` also would require + enrollment relations neither students nor courses could be instantiated. Java provides no primitive to create a single `Course`, a single `Student` and a relation between them in an atomic step.

Alloy: Model Checking Multiplicity Invariants Alloy [11] is not a programming language but a specification language. It does however poses a means of specifying an atomic update that does exactly what statements cannot do: create a student, a course and a relation between those in a single step. The mechanism for specifying updates is the Z notation [19]: a relation between a pre and post state. To check whether the invariants can be broken by updates bounded model checking is done.

Booster: Multiplicity-Safe by Runtime Availability Booster [6] is a language which also uses the Z notation for specifying updates. In Booster multiplicity invariants are ensured by runtime availability of updates: if performing an update would mean violating a constraint then the update operation is not available. The invariants are ensured, but the programmer might specify updates that are available in less cases than what he expects.

Our Requirements We would like to combine the static guarantees one gets with multiplicity-safe statements with the expressivity from the Z notation. Compared to Booster we would like to move the multiplicity checks to compile time instead of runtime. It should still be able for updates to be unavailable at runtime (for example trying to delete a course while there are still students enrolled), but that unavailability should be made explicit by the programmer. The type system should check that in all other cases the operation is available, or give a type error.

6. Related Work

Our work builds on research in different fields: language constructs for relations, navigating and querying relations and multiplicities. Specific differences with our work are highlighted per article.

Languages with first-class relations The Rumer language by Balzer has first-class relations [2, 3]. It features first-class relations with named roles and queries. Rumer provides reactive queries as well as imperative code. It has cardinalities specified in constraints and implements binary relationships. Our approach on the other hand does not support imperative code, has multiplicities as part of the type system and features relations of all degrees.

Classages is a language that also features relations [14]. Classages is targeted at modelling the interactions and interaction life span between objects. It features static and dynamic relations, bidirectional relations and multiplicities. Our approach has in common that it has bidirectional relations but we are focused on modeling data instead of interactions.

Pearce and Noble extended Java with first-class relationships using aspects [17]. Relations are modeled as external tuples and objects are agnostic to relations they are in. Their approach to behavioural changes of objects based on their relations should be implemented by aspects, externally. Our approach is the opposite, entities know what relations they participate in. This allows specifying relation dependent behaviour in derivations.

RelJ is first-class relationship extension to Java by Biermann and Wren [4, 23]. In their approach they support relationships as first-class citizens. The relations are also modeled as tuples, where the roles have a position in the tuple but no name. In our approach the roles are named and unordered; allowing navigation based on roles. Their relations are binary and one-directional. In the technical report they also sketch an extension with named roles [4]. In this sketched extension relations can have any arity and support bidirectional navigation.

Nelson implemented first-class relationships in Java [16]. This is a library and not a language extension. Mutable sets of tuples are used as first-class constructs to model relations. Without specific language constructs it does not supply additional semantics for relations and thus cannot provide additional static checking.

Languages with non first-class relations In 1987 Rumbaugh was the first to add relations to a language [18]. His approach is preprocessor based and dynamic. It does not have relations as first-class citizens.

In 1991 a relationship mechanism for a Strongly Typed Object-Oriented Database Programming language introduced statically typed relations as part of a language [1]. The paper explains the data model definition and transactions. It does however not explain in detail how querying or navigation is done.

WebDSL introduced inverse properties which inspired the inverses [21]. Refer to Section 3 for details.

Queries of relations in Object-Oriented languages The Java Query Language (JQL) adds queries to Java [22]. There is no additional support for relations, so navigation uses value-based joins like in SQL. LINQ also uses value-based joins [15]. These approaches are in the left column of the design space (Section 3). In contrast, our navigation is based on the role names of relations.

Multiplicities in programming languages In Content over Container: Object-Oriented Programming with multiplcities Steimann adds multiplicity annotations to Java in order to remove the Collection containers [20]. Refer to Section 2 for details.

Finally the ideas for this paper were presented in the ACM Student Research Competition [8]. The design space analysis and formal semantics of the language are new to this paper. Also the syntax changed as a result of the design-space analysis.

7. Conclusion

Unification and generalization of relations led to a new data modeling and navigation language. This goes hand in hand with native multiplicities. Both the relations aspect and the native multiplicities aspect lead to more a more concise definition and navigation of relationships; removing maintenance of reference consistency, removing collection classes and providing single identifier navigation by inverses and shortcuts.

References

[1] Albano, A., Ghelli, G., Orsini, R.: A relationship mechanism for a strongly typed object-oriented database programming language. In: VLDB. pp. 565–575 (1991)

[2] Balzer, S.: Rumer: a Programming Language and Modular Verification Technique Based on Relationships. Ph.D. thesis, ETH, Zürich (2011)

[3] Balzer, S., Gross, T.R., Eugster, P.: A relational model of object collaborations and its use in reasoning about relationships. In: ECOOP. pp. 323–346 (2007)

[4] Bierman, G.M., Wren, A.: First-class relationships in an object-oriented language. In: ECOOP. pp. 262–286 (2005)

[5] Chen, P.P.: The entity-relationship model - toward a unified view of data. tods 1(1), 9–36 (1976)

[6] Davies, J., Welch, J., Cavarra, A., Crichton, E.: On the generation of object databases using booster. In: ICECCS. pp. 10–pp. IEEE (2006)

[7] Halpin, T.: Object-role modeling (orm/niam). In: Handbook on architectures of information systems, pp. 81–103. Springer (2006)

[8] Harkes, D.: Relations: a first class relationship and first class derivations programming language. In: AOSD. pp. 9–10 (2014)

[9] Harkes, D., Visser, E.: Unifying and generalizing relations in role-based data modeling and navigation. In: SLE. pp. 241–260 (2014)

[10] Hemel, Z., Groenewegen, D.M., Kats, L.C.L., Visser, E.: Static consistency checking of web applications with WebDSL. JSC 46(2), 150–182 (2011)

[11] Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM) 11(2), 256–290 (2002)

[12] Jacobson, I., Booch, G., Rumbaugh, J.E.: The unified software development process - the complete guide to the unified process from the original designers. Addison-Wesley object technology series, Addison-Wesley (1999)

[13] Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: OOPSLA. pp. 444–463 (2010)

[14] Liu, Y.D., Smith, S.F.: Interaction-based programming with classages. In: OOPSLA. pp. 191–209 (2005)

[15] Meijer, E., Beckman, B., Bierman, G.M.: Linq: reconciling object, relations and xml in the .net framework. In: sigmod. p. 706 (2006)

[16] Nelson, S., Noble, J., Pearce, D.J.: Implementing first-class relationships in java. Proceedings of RAOOL 8 (2008)

[17] Pearce, D.J., Noble, J.: Relationship aspects. In: AOSD. pp. 75–86 (2006)

[18] Rumbaugh, J.E.: Relations as semantic constructs in an object-oriented language. In: OOPSLA. pp. 466–481 (1987)

[19] Spivey, J.M., Abrial, J.: The Z notation. Prentice Hall Hemel Hempstead (1992)

[20] Steimann, F.: Content over container: object-oriented programming with multiplicities. In: OOPSLA. pp. 173–186 (2013)

[21] Visser, E.: WebDSL: A case study in domain-specific language engineering. In: GTTSE. pp. 291–373 (2007)

[22] Willis, D., Pearce, D.J., Noble, J.: Efficient object querying for java. In: ECOOP. pp. 28–49 (2006)

[23] Wren, A.: Relationships for object-oriented programming languages. University of Cambridge, Computer Laboratory, Technical Report 702(UCAM-CL-TR-702) (November 2007)