Unifying and Generalizing Relations in Role-Based Data Modeling and Navigation

Daco Harkes and Eelco Visser

Delft University of Technology, The Netherlands
d.c.harkes@student.tudelft.nl, visser@acm.org

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.

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 [6], UML [10] 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 [19] 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).

B. Combemale et al. (Eds.): SLE 2014, LNCS 8706, pp. 241–260, 2014.

```
class Student { }

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

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

Fig. 1. Multiplicity annotations in Java

- 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).
- A formal definition of the type system (Section 5) and operational semantics (Section 6) of this language.

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 [19]:

- 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 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 [19]. 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()); }
}
```

Fig. 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. In Section 5 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. Figures 3 and 4 summarize the complete design space in tabular form emphasizing its regularities. From this analysis a new data modeling language emerges which unifies and generalizes the various approaches to modeling relations.

In all our examples we assume the language to have native multiplicities instead of using collections that would be needed in a plain OO approach. The running example data model defines `Student`s who are enrolled in `Course`s, sometimes via a first-class `Enrollment` relation. For the sake of the example, students can be enrolled in zero or more courses (`*` multiplicity), and courses should have at least one student (`+` multiplicity). In the example expressions we use `Student` 'bob' and `Course` 'math'. For each point in the design space we give a type graph diagram describing the data model, a textual specification of the data model, and expressions for querying the model. For the expressions we use `=>` to express the result of evaluation.

3.1 Overview

Before discussing each point in the design space (Figures 3 and 4) individually, we first introduce the categories represented by the columns and rows.

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 `c` we use the navigation expression `c<-(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 [20] 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 (J and K), but that is not particularly elegant.

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 (see E). 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.

3.2 Detailed Description of Points in Design Space

We discuss each of the points A to K in the design space (Figures 3 and 4).

Object-Oriented (A, B and C). There are multiple patterns for modeling relations in objected-oriented languages [16]. As mentioned before, we replace collections by multiplicities and do not consider patterns with redundant references for bidirectional navigation. Three basic patterns remain: reference (A), relation tuple (B), and relation class (C), which we assume to be familiar to the reader. It is noteworthy that a language extension is not required for the representation

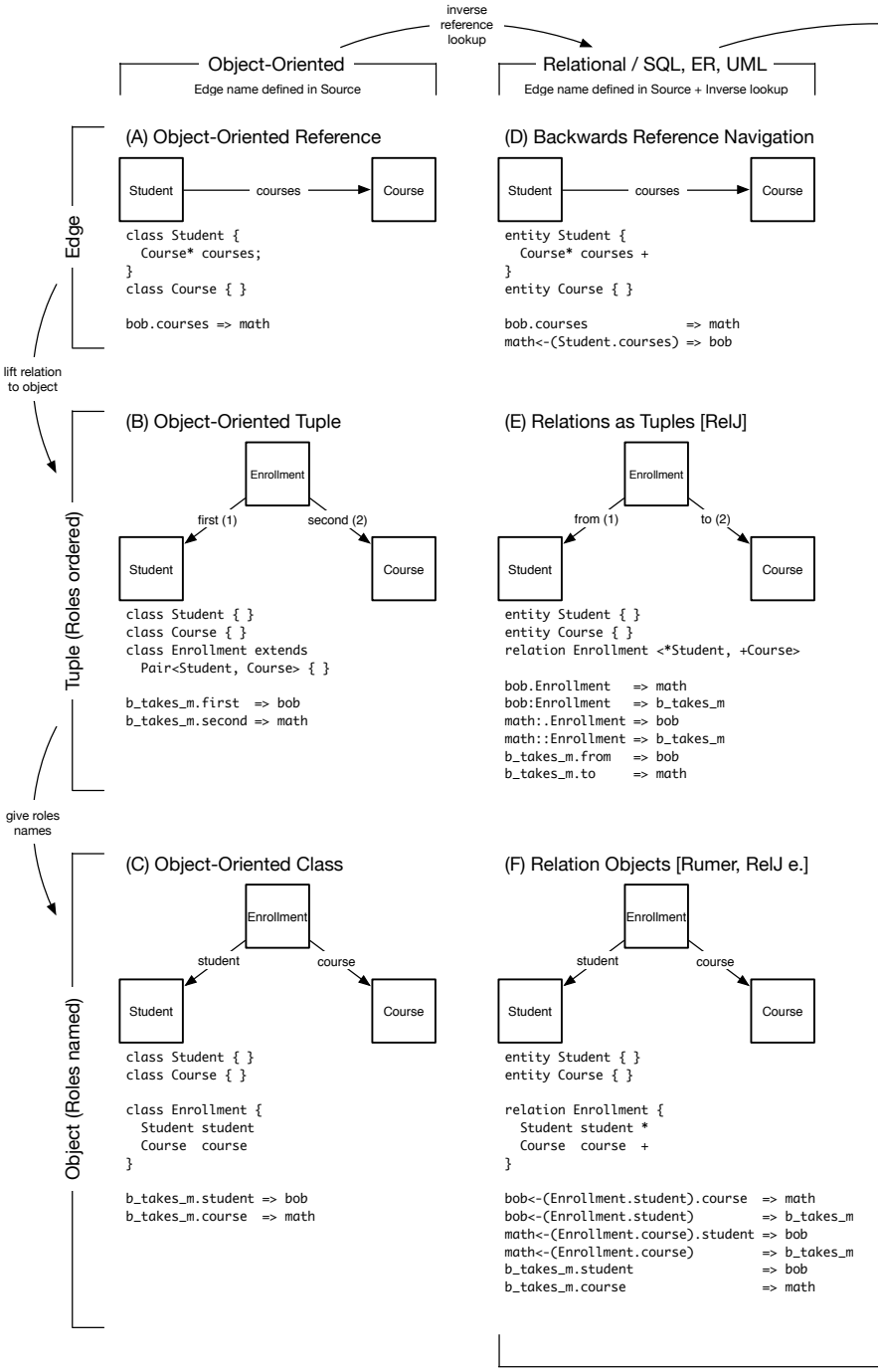

Fig. 3. Design space of relations in data modeling and navigation (part 1)

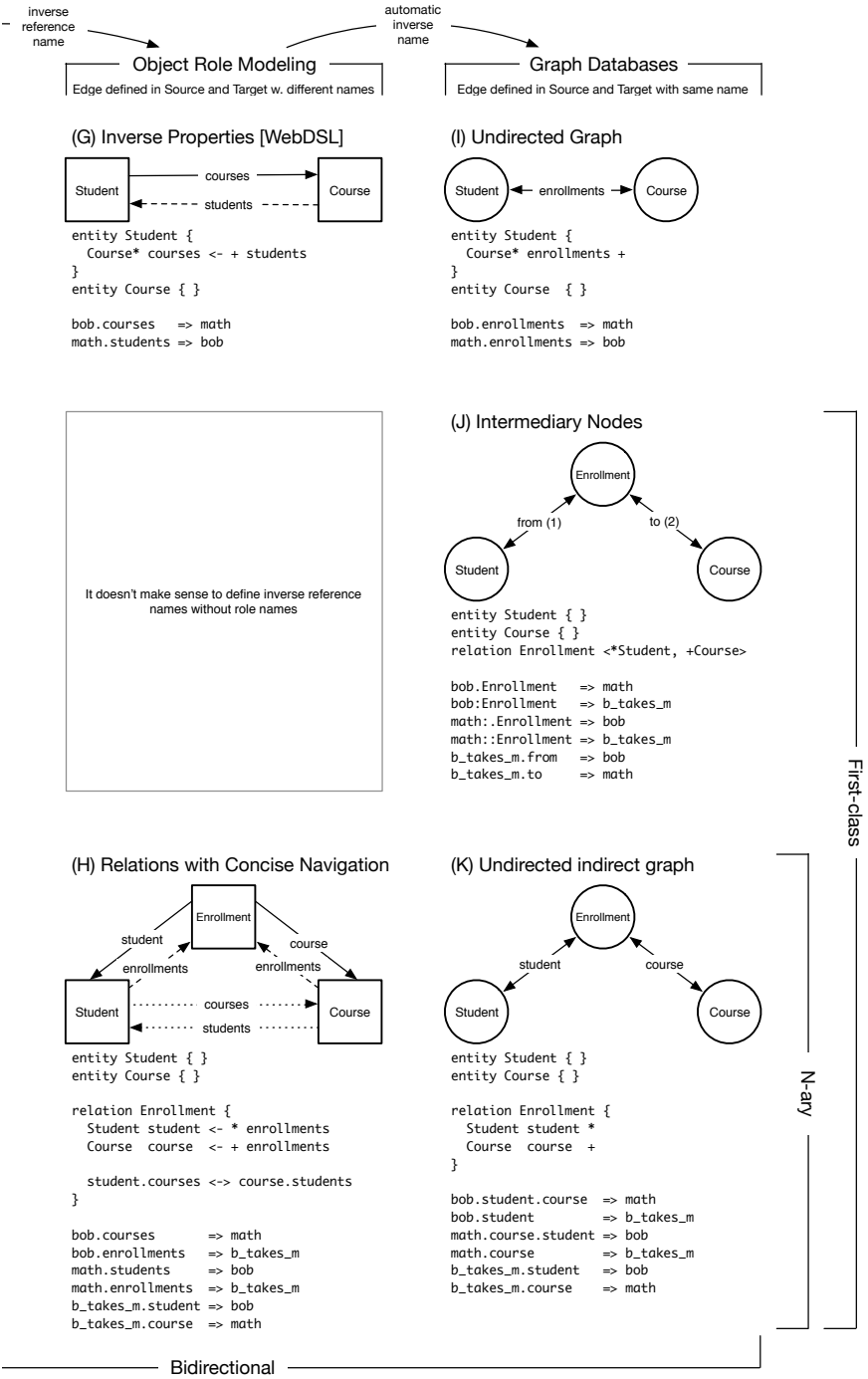

Fig. 4. Design space of relations in data modeling and navigation (part 2)

```
class Student { }
class Course { }
relationship Enrollment (Student, Course) { int grade; }

bob.Enrollment       // bob's courses
bob:Enrollment       // Enrollment-type relation objects
bob:Enrollment.grade
b_takes_m.from       // bob
b_takes_m.to         // math
```

Fig. 5. First-class citizen tuple based relations in RelJ [4]

of first-class relations. The term first-class is sometimes used for having a dedicated language construct, but a dedicated language construct is not required for adding attributes to relations or letting relations participate in other relations. First-class relations based on tuples (B) have been implemented as a Java library [15].

Backwards reference navigation (D). If we extend an object-oriented language with facilities for backwards reference lookup (`o<-(T.r)`) we can use a single reference for bidirectional navigation. Note that in this case the object graph is identical to the single reference pattern (A).

Relation as Tuples (E). The RelJ Java extension lifts relations to tuple objects [4]. In RelJ different operators are used to disambiguate between different navigation operations (Figure 5). RelJ provides no facilities for bidirectional navigation. However, that is not a conceptual limitation. Adding two operators (`:.` and `::`) would allow backward navigation, as suggested in (E). While this is theoretically extensible to relations with more than two participants, it requires adding new operators for each participant.

Relation Objects (F). Naming roles allows usable extension to n-ary relations. This is the model used by Rumer [2,3] as illustrated in Figure 6. 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 [22] adds names to roles, as illustrated in Figure 7, 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.

Inverse Properties (G) WebDSL [20] supports bidirectional navigation without a verbose syntax for inverse lookups by means of *inverse properties* [9] as illustrated in Figure 8. Explicit names on both sides of an association simplifies navigation to just following named references. However, these names have to be

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

Fig. 6. 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 students
```

Fig. 7. Ternary relation extension proposal for RelJ [22]

defined in both the source and target class. In (G) we have normalized this to a single property definition with two names; the second name is used for the backwards reference from target to source.

Concise Relations (H). Combining the advantages of (F) and (G), we arrive at our proposal for a unified and generalized approach to modeling relations (H). 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 (G), 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 (G), these declarations do not introduce attributes in the participant classes, but rather shortcuts. For example, `bob.courses` is a shortcut for `bob.enrollments.course`. This approach naturally extends to n-ary relations, as illustrated in Figure 9.

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

Fig. 8. Inverse properties in WebDSL

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

  student.courses <-> course.students
  student.tutors  <-> tutor.students
  course.tutors   <-> tutor.courses
}
```

Fig. 9. Ternary relation with concise navigation (H) (this paper)

Undirected Graphs (I, J, K). Graph databases also feature three relation patterns. The simple edge (I), adding an intermediary node without role names (J), and an intermediary node with role names (K). Since without edge names, edge directionality does not matter (J) is equivalent to (E). So we will only cover (I) and (K).

The simple edge (I) cannot be used to model asymmetric same type relations. Asymmetric relations of the different types can be disambiguated by the type one starts navigating from, but if both participants have the same type their role is ambiguous. Disambiguation can be done through indirection (I or K). With indirection (K) navigation from participant to participant is navigating two edges. With undirected edges role names cannot be reused with different relations concerning the same entity. Consider adding another relation where `Course` also participates as `course`. `math.course` now becomes ambiguous. The language could then be extended with the type of the node navigating to, but this is equivalent to the backwards reference navigation: naming the edge and the type on the other side. So that would bring us back at (F).

It seems there is a fundamental trade-off between undirected and directed graphs when considering reference names. The directed graph (column two) requires an extra identifier (the target type) to navigate edges backwards. To get rid of this extra identifier we can automatically define the edge name on both sides. This is gets us to the undirected graph (column four). In undirected graphs we have ambiguities. Adding an extra identifier (the target type) to disambiguate brings us back at the directed graphs.

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 point (H) in the design space. In this section we discuss two extensions of the basic idea of (H) and the grammar of the language. In the next sections we give a formal definition of the type system and operational semantics.

```
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
}
```

Fig. 10. Expansion of concise relation definition

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

Fig. 11. Relations language with derivation

Concise Definition of Relations. While navigation according to (H) is very concise, the definition of a relation is somewhat verbose due to the introduction of names for each of the arrows in the diagram. 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 designers workbench Spoofax [11]. The prototype is publicly available.[1] The type system and semantics described in the next sections matches those of the prototype.

[1] `https://github.com/metaborg/relationstagv0.2.0`

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

Fig. 12. The grammar of the relations language

5 Type System

Our language features static typing. Everything in the language has both a *type* and a *multiplicity*. These are defined orthogonally.

Meta variables. In the the static and dynamic semantic rules we use a meta variables for looking up definitions on usage sites.

$$
\begin{aligned}
\mathcal{P} \in \mathit{Program} &: \mathit{EntityMap} \times \mathit{Expr} \\
\mathcal{E} \in \mathit{EntityMap} &: \mathit{EntityName} \to \mathit{AttributeMap} \times \mathit{InverseMap} \times \mathit{RoleMap} \\
\mathcal{A} \in \mathit{AttributeMap} &: \mathit{AttrName} \to \mathit{PrimitiveType} \times \mathit{Multiplicity} \times \mathit{Expr} \\
\mathcal{I} \in \mathit{InverseMap} &: \mathit{InverseName} \to \mathit{EntityName} \times \mathit{RoleName} \times \mathit{RoleName} \\
\mathcal{R} \in \mathit{RoleMap} &: \mathit{RoleName} \to \mathit{EntityName} \times \mathit{Multiplicity}
\end{aligned}
$$

A program $\mathcal{P}$ is a tuple, $(\mathcal{E}, e)$, where $\mathcal{E}$ is a map from entity (and relation) names to entity definitions and e is the main expression.

Entity definitions are triples $(\mathcal{A}, \mathcal{I}, \mathcal{R})$, where $\mathcal{A}$ is a map from attribute names to attribute definitions, $\mathcal{I}$ is a map of inverse names to their origin and $\mathcal{R}$ is a map from role names to role definitions. Both entities and relations define entities. We refer to an entity t's attribute, inverse and role map as $\mathcal{A}_t$, $\mathcal{I}_t$ and $\mathcal{R}_t$ respectively.

Attribute definitions are triples (p, m, e), where p is the primitive type, m is the multiplicity and e is the optional derivation expression. If e has no derivation expression it is equal to `nil`. Role definitions are tuples (t, m), where t is an entity name and m is a multiplicity. Inverse (and shortcut) definitions are triples (t, r_1, r_2) where r_1 and r_2 are roles in entity t. The inverse map definition is best explained by example:

```
entity Enrollment {
  Student student <– * enrollment
  Course  course  <– + enrollment
  student.courses <–>  course.students
}
```

$$
\begin{aligned}
\mathcal{I}_{Student} : \text{'enrollment'} &\rightarrow \text{'Enrollment'} \times \text{'student'} \times \mathit{nil} \\
\text{'courses'} &\rightarrow \text{'Enrollment'} \times \text{'student'} \times \text{'course'} \\
\mathcal{I}_{Course} : \text{'enrollment'} &\rightarrow \text{'Enrollment'} \times \text{'course'} \times \mathit{nil} \\
\text{'students'} &\rightarrow \text{'Enrollment'} \times \text{'course'} \times \text{'students'}
\end{aligned}
$$

The inverses of roles are mapped back to the role in the relation they are the inverse of. In this case r_2 is `nil`. The shortcut is translated to two records, one for both participant types. The inverse maps are used as the backwards reference navigation mechanism.

Lastly, to simplify static and dynamic semantics we transform the shortcut expressions to an inverse and a role expression by the transformation rule:

$$
\frac{e : t_1 \qquad \mathcal{I}_{t1}(i_1) = (t_2, r_1, r_2) \qquad \mathcal{I}_{t1}(i_2) = (t_2, r_1, \text{nil})}{e \;.\; i_1 \;\rightarrow\; e \;.\; i_2 \;.\; r_2}
$$

Types. There are two type sorts: p (*primitive types*) and t (*entity types*). All attributes are primitive types. Entities and relations define entity types. Roles, inverses and shortcuts in a relation are entity types.

Most typing rules are straightforward, so we only cover the rules that are non-standard. The aggregation rule (Aggr) is interesting. Since multiplicities are encoded orthogonally the aggregation functions are of type int $\rightarrow$ int. The multiplicity operators choice and concatenate work with any type. They only check whether both operands have the same type and propagate the type (Mult).

With roles and inverses one can conceptually navigate over the type graph defined by the entities and relations. The type of a navigation expression is naturally the place where one ends up in the model after navigating. When navigating from a relation to a participant the type is the participant's type (RoleNav). When navigating from a participant to a relation, by an inverse, we find the type of the relation by looking up the inverse definition (InvNav).

Multiplicities. For multiplicities there are two notational conventions: single characters from the concrete syntax and ranges. We use the ranges notation in the multiplicity rules as it gives us access to the upper and lower bounds directly.

Binary operators mimic maybe-Monad behaviour for zero or one values: a maybe value as input for the computation returns a maybe value as output. Taking the Cartesian product between the bags of values and applying the operation to each pair provides this behaviour. The multiplicity range is expressed as taking the minimum of both lower bounds and the maximum of the upper bounds (BinOp). The division and modulo operators exhibit slightly different behaviour (DivOp). Since dividing by zero has no result, at least one value in

$$\frac{c \in \{\text{true}, \text{false}\}}{c : \text{boolean}}\ [\text{Bool}] \qquad \frac{}{literalInt : \text{int}}\ [\text{Int}] \qquad \frac{}{literalString : \text{string}}\ [\text{Str}] \qquad \frac{}{\theta \vdash \text{this} : \theta}\ [\text{This}]$$

$$\frac{\oplus \in \{+, -, *, /, \%\} \quad e_1 : \text{int} \quad e_2 : \text{int}}{e_1 \oplus e_2 : \text{int}}\ [\text{Math}] \qquad \frac{e_1 : \text{string} \quad e_2 : \text{string}}{e_1 + e_2 : \text{string}}\ [\text{Conc}]$$

$$\frac{\oplus \in \{\&\&, ||\} \quad e_1 : \text{boolean} \quad e_2 : \text{boolean}}{e_1 \oplus e_2 : \text{boolean}}\ [\text{AndOr}] \qquad \frac{e : \text{boolean}}{!\ e : \text{boolean}}\ [\text{Not}]$$

$$\frac{\oplus \in \{>, >=, <, <=\} \quad e_1 : t \quad e_2 : t \quad t \in \{\text{int}, \text{string}\}}{e_1 \oplus e_2 : \text{boolean}}\ [\text{Cmp}]$$

$$\frac{e_1 : t \quad e_2 : t \quad \oplus \in \{==, !=\}}{e_1 \oplus e_2 : \text{boolean}}\ [\text{Eq}] \qquad \frac{e_1 : \text{boolean} \quad e_2 : t \quad e_3 : t}{e_1\ ?\ e_2\ ":"\ e_3 : t}\ [\text{Cond}]$$

$$\frac{e : \text{int} \quad f \in \{\text{avg}, \text{min}, \text{max}, \text{sum}\}}{f(e) : \text{int}}\ [\text{Aggr}] \qquad \frac{e : \text{boolean} \quad f \in \{\text{conj}, \text{disj}\}}{f(e) : \text{boolean}}\ [\text{Logic}]$$

$$\frac{e : \_}{\text{count}(e) : \text{int}}\ [\text{Count}] \qquad \frac{e_1 : t \quad e_2 : t \quad \oplus \in \{<+, ++\}}{e_1 \oplus e_2 : t}\ [\text{Mult}]$$

$$\frac{e : t \quad \mathcal{A}_t(a) = (p, \_.\_)}{e\ .\ a : p}\ [\text{Attr}] \qquad \frac{e : t_a \quad \mathcal{A}_t(a) = (t_a, \_)}{t\ [\ a == e\ ] : t}\ [\text{Lookup}]$$

$$\frac{e : t \quad \mathcal{R}_t(r) = (t_r, \_)}{e\ .\ r : t_r}\ [\text{RoleNav}] \qquad \frac{e_1 : t_1 \quad \mathcal{I}_{t1}(i) = (t_2, \_, \text{nil})}{e\ .\ i : t_2}\ [\text{InvNav}]$$

Fig. 13. Type rules

$$\frac{c \in \{\text{true}, \text{ false}, \text{ false}, \ Int, \ String\}}{c \sim [1, 1]}\ [\text{Const}]$$

$$\frac{\oplus \in \{+, -, *, \&\&, ||, >, >=, <, <=, ==, !=\} \quad e_1 \sim [l_1, u_1] \quad e_2 \sim [l_2, u_2]}{e_1 \oplus e_2 \sim [min(l_1, l_2), max(u_1, u_2)]}\ [\text{BinOp}]$$

$$\frac{\oplus \in \{/, \%\} \quad e_1 \sim [\_, u_1] \quad e_2 \sim [\_, u_2]}{e_1 \oplus e_2 \sim [0, max(u_1, u_2)]}\ [\text{DivOp}]$$

$$\frac{e_1 \sim [l_1, 1] \quad e_2 \sim [l_2, u_2] \quad e_3 \sim [l_3, u_3] \quad m = [min(l_1, l_2, l_3), max(u_2, u_3)]}{e_1\ ?\ e_2\ ":"\ e_3 \sim m}\ [\text{Cond}]$$

$$\frac{e \sim m}{!\ e \sim m}\ [\text{Not}] \qquad \frac{f \in \{\text{avg}, \text{min}, \text{max}, \text{conj}, \text{disj}\} \quad e \sim [l, \text{n})}{f(e) \sim [l, 1]}\ [\text{Aggr}]$$

$$\frac{f \in \{\text{sum}, \text{count}\}}{f(e) \sim [1, 1]}\ [\text{Aggr2}] \qquad \frac{e_1 \sim [0, u_1] \quad e_2 \sim [l_2, u_2]}{e_1 <+\ e_2 \sim [l_2, max(u_1, u_2)]}\ [\text{Choice}]$$

$$\frac{e_1 \sim [1, u_1]}{e_1 <+\ e_2 \sim [1, u_1]}\ [\text{Choice2}] \qquad \frac{e_1 \sim [l_1, \_] \quad e_2 \sim [l_2, \_]}{e_1 ++\ e_2 \sim [max(l_1, l_2), n)}\ [\text{Concat}]$$

$$\frac{e \sim [l_1, u_1] \quad \mathcal{A}_{t_e}(a) = (\_, [l_2, 1], \_)}{e\ .\ a \sim [min(l_1, l_2), u_1]}\ [\text{Attr}] \qquad \frac{}{t\ [\ a == e\ ] \sim [0, n)}\ [\text{Lookup}]$$

$$\frac{e : t \quad e \sim m \quad \mathcal{R}_t(r) = (\_, \_)}{e\ .\ r \sim m}\ [\text{RoleNav}] \qquad \frac{e_1 : t_1 \quad \mathcal{I}_{t1}(i) = (t_2, r, \text{nil}) \quad \mathcal{R}_{t2}(r) = (\_, [l_2, u_2])}{e\ .\ i \sim [min(l_1, l_2), max(u_1, u_2)]}\ [\text{InvNav}]$$

Fig. 14. Multiplicity rules

$$\frac{a = (\_, [\_, 1], \text{nil})}{\vdash a} \quad \text{[AttrDec]}$$

$$\frac{a = (p, [l_1, 1], e) \qquad e : p \qquad e \sim [l_2, 1] \qquad l_1 \leq l_2}{\vdash a} \quad \text{[AttrDec2]}$$

$$\frac{r = (t, m) \qquad \mathcal{E}(t) = (\_, \_)}{\vdash r} \quad \text{[RoleDec]}$$

$$\frac{i = (t, r_1, \text{nil}) \qquad \mathcal{R}_t(r_1) = (\_, \_)}{\vdash i} \quad \text{[InvDec]}$$

$$\frac{i = (t, r_1, r_2) \qquad \mathcal{R}_t(r_1) = (\_, \_) \qquad \mathcal{R}_t(r_2) = (\_, \_)}{\vdash i} \quad \text{[ShortcutDec]}$$

$$\frac{\theta' = t \qquad \forall a \in dom(\mathcal{A}_t) : \theta' \vdash a \qquad \forall r \in dom(\mathcal{R}_t) : \theta' \vdash r \qquad \forall i \in dom(\mathcal{I}_t) : \theta' \vdash i}{\vdash t} \quad \text{[EntityDec]}$$

$$\frac{\theta' = \bot \qquad \forall t \in dom(\mathcal{E}) : \theta' \vdash t \qquad \theta' \vdash e : \_ \qquad \theta' \vdash e \sim \_}{\vdash (\mathcal{E}, e)} \quad \text{[ProgramDec]}$$

Fig. 15. Attribute, role, inverse, shortcut, entity and program well-formedness

both operands might still result in no answer. Instead of throwing a division by zero exception zero answers are given for any denominator equal to zero.

The CHOICE operator chooses at runtime the left expression if it has a result, and otherwise the right expression. The multiplicity is defined as the maximum of both upper and lower bound, except if the left lower bound is one. Then we know that the left expression will always be chosen. Note that it does not make sense to use the choice operator in that case, because the right expression will be dead code. The CONCAT operator combines the results of both expressions. This means that we might always have more than one value at runtime; thus the upper bound is n. The lower bound is the maximum of both.

Attributes are allowed to be either [0,1] or [1,1]. In the first case attribute access decreases the lower bound to zero, as the attribute might not be set (ATTR). A role always has exactly one value, so role navigation leaves multiplicity intact (ROLENAV). Navigation to relations entities participate in behaves like a SQL join between the input expression entities and the relation. Like binary operators this means taking the lowest lower bound and the highest upper bound.

Well-formedness. Programs are well-formed if they satisfy the rules in Figure 15. Attributes are only allowed to have a multiplicity of at most one, their type has to be primitive (which is enforced by the syntax definition already) and if a derivation is specified, it should be of the correct type and its multiplicity should fit inside the target range. Role declarations are well-formed if the entity playing the role exists in the entity map. Inversions are well-formed if the role exists in the entity of which they are the inverse and shortcuts are well-formed if both roles exist in the entity. Entity definitions are well-formed if all their attributes, inverses and roles are well-formed and a program is well-formed if all its entities and the main expression are well-formed. We only consider well-formed programs.

6 Dynamic Semantics

We specify evaluation rules for a big-step semantics. We use the I-MSOS notational style, which implicitly propagates stores if they are not mentioned [14].

Stores. In order to evaluate a program an entity store Σ and relation store Δ must be passed; our language is a data modeling and navigation language and does not provide facilities to add, edit or remove data. Expression in addition get passed a this-reference θ.

$$\Sigma, \Delta \vdash p \Downarrow v \quad \text{(Evaluation of program)}$$
$$\Sigma, \Delta, \theta \vdash e \Downarrow v \quad \text{(Evaluation of expressions)}$$

The entity store corresponds to the usual heap: a map from object references to a map from attribute names to their values. The relation store is used for storing all relations between entities. It is a map from relation name, relation object reference and role name to the reference of the object playing this role. The this-reference is a single reference to an object.

$$\begin{aligned}
\Sigma &\in EntityStore : Reference \to AttributeStore \\
&AttributeStore : AttrName \to Value \\
\Delta &\in RelationStore : EntityName \times Reference \times RoleName \to Reference \\
\theta &\in ThisReference : Reference
\end{aligned}$$

Store well-formedness. Figure 16 describes what it means means for these stores to be well-formed. The entity store is well-formed if all the entities in it are well-formed. An entity is well-formed if (1) all records in its attribute store are well-formed, (2) all its required, non-derived attributes have been set (3) all its roles have a value and (4) the number of relation records, that point to it for a certain role that he plays, is within the multiplicity range specified for that role.

An attribute record is well-formed if it has a value of the correct type. The relation store is well-formed if all its records are well-formed. A relation record is well-formed if its references point to entities. Finally the this-reference is well-formed if it points to an entity. We assume a well-formed entity and relation stores for evaluation.

Evaluation rules All the evaluation rules have a specific form: they operate on bags. Expressions can return any number of values, modeling this with bags is a natural choice. A nice example of this is the rule for binary operations (BinOp). The left and right expressions evaluate to a bag of values, the Cartesian product of these bags is taken and on each pair of values the operator is applied. For single values a normal computation is performed, for maybe values a maybe computation and for many values a Cartesian product computation. Most evaluation rules follow this pattern.

Aggregation operations are defined for at least a single value (Aggr) and for empty lists there is predefined behaviour (Aggr2 and Sum). Choice returns

$$\frac{\forall(ref \to astore) \in \Sigma : \vdash (ref \to astore)}{\vdash \Sigma} \quad \text{[EntityStore]}$$

$$\frac{\begin{array}{l} ref : t \\ \forall(a \to v) \in astore : \ ref \vdash (a \to v) \\ \forall(a \to p, [1,1], \_) \in \mathcal{A}_t : \ astore(a) = \_ \\ \forall(r \to \_, \_) \in \mathcal{R}_t : \Delta(t, ref, r) = \_ \\ \forall(i \to t_2, r_2, \text{nil}) \in \mathcal{I}_r : \\ \quad (|\{v \mid \Delta(t_2, \_, r_2) = v\}| = m \quad \mathcal{R}_{t2}(r_2) = (\_, [l,u]) \quad l \le m \le u) \end{array}}{\vdash (ref \to astore)} \quad \text{[EntityRecord]}$$

$$\frac{e : t \quad \mathcal{A}_t(a) = (t_a, \_, \_) \quad v : t_a}{e \vdash a \to v} \quad \text{[AttrRecord]}$$

$$\frac{\forall(t\ v_1\ r \to v_2) \in \Delta : \vdash (t\ v_1\ r \to v_2)}{\vdash \Delta} \quad \text{[RelationStore]}$$

$$\frac{v_1 : t \quad \Sigma(v_1) = \_ \quad \mathcal{R}_t(r) = (t_2, \_) \quad v_2 : t_2 \quad \Sigma(v_2) = \_}{\vdash t\ v_1\ r \to v_2} \quad \text{[RelationRecord]}$$

$$\frac{\Sigma(\theta) = \_}{\vdash \theta} \quad \text{[ThisReference]}$$

Fig. 16. Store well-formedness

$$\frac{c \text{ is constant}}{c \Downarrow \{|\ c\ |\}} \quad \text{[Const]}$$

$$\frac{}{\theta \vdash \text{this} \Downarrow \{|\ \theta\ |\}} \quad \text{[This]}$$

$$\frac{\begin{array}{l} \oplus \in \{+, -, *, \&\&, ||, >, \\ \quad >=, <, <=, ==, !=\} \\ e_1 \Downarrow V_1 \quad e_2 \Downarrow V_2 \end{array}}{\begin{array}{l} e_1 \oplus e_2 \Downarrow \{|\ v_1 \oplus v_2 \mid \\ \quad v_1 \in V_1,\ v_2 \in V_2\ |\} \end{array}} \quad \text{[BinOp]}$$

$$\frac{e_1 \Downarrow V_1 \quad e_2 \Downarrow V_2 \quad \oplus \in \{/, \%\}}{\begin{array}{l} e_1 \oplus e_2 \Downarrow \{|\ v_1 \oplus v_2 \mid v_2 \mathrel{!}= 0, \\ \quad v_1 \in V_1,\ v_2 \in V_2\ |\} \end{array}} \quad \text{[Div]}$$

$$\frac{e \Downarrow V}{!\ e \Downarrow \{|\ \neg\ v \mid v \in V\ |\}} \quad \text{[Not]}$$

$$\frac{e_1 \Downarrow V_1 \quad e_2 \Downarrow V_2 \quad e_3 \Downarrow V_3}{\begin{array}{l} e_1\ ?\ e_2 : e_3 \Downarrow \{|\ v_1\ ?\ v_2 : v_3 \mid v_1 \in V_1, \\ \quad v_2 \in V_2,\ v_3 \in V_3\ |\} \end{array}} \quad \text{[Cond]}$$

$$\frac{\begin{array}{l} f \in \{\text{avg}, \text{min}, \text{max}, \text{conj}, \text{disj}, \text{sum}\} \\ e \Downarrow V \quad |V| \ge 1 \end{array}}{f(e) \Downarrow \{|\ f(V)\ |\}} \quad \text{[Aggr]}$$

$$\frac{\begin{array}{l} f \in \{\text{avg}, \text{min}, \text{max}, \text{conj}, \text{disj}\} \\ e \Downarrow \emptyset \end{array}}{f(e) \Downarrow \emptyset} \quad \text{[Aggr2]}$$

$$\frac{e \Downarrow \emptyset}{\text{sum}(e) \Downarrow \{|\ 0\ |\}} \quad \text{[Sum]}$$

$$\frac{e \Downarrow V}{\text{count}(e) \Downarrow \{|\ |V|\ |\}} \quad \text{[Count]}$$

$$\frac{e_1 \Downarrow V_1 \quad e_2 \Downarrow V_2}{e_1 <+\ e_2 \Downarrow (V_1 \mathrel{!}= \emptyset)\ ?\ V_1 : V_2} \quad \text{[Choice]}$$

$$\frac{e_1 \Downarrow V_1 \quad e_2 \Downarrow V_2}{e_1 +\!+\ e_2 \Downarrow V_1 \cup V_2} \quad \text{[Concat]}$$

$$\frac{e \Downarrow V \quad e : t \quad \mathcal{A}_t(a) = (\_, \_, \text{nil})}{\Sigma \vdash e\ .\ a \Downarrow \{|\ \Sigma(v)(a) \mid v \in V\ |\}} \quad \text{[Attr]}$$

$$\frac{\begin{array}{l} e \Downarrow V \quad e : t \quad \mathcal{A}_t(a) = (\_, \_, e_2) \\ V_2 = \{|\ v_2 \mid (\theta' \vdash e_2 \Downarrow \{v_2\}),\ \theta' \in V|\} \end{array}}{e\ .\ a \Downarrow V_2} \quad \text{[At2]}$$

$$\frac{e \Downarrow V \quad e : t}{\Delta \vdash e\ .\ r \Downarrow \{|\ \Delta(t, v, r) \mid v \in V\ |\}} \quad \text{[RoleNav]}$$

$$\frac{\begin{array}{l} e \Downarrow V \quad e : t \quad \mathcal{I}_t(i) = (t, r, \text{nil}) \\ V_2 = \{|\ v_2 \mid \Delta(t, v_2, r) = v,\ v \in V|\} \end{array}}{\Delta \vdash e\ .\ i \Downarrow V_2} \quad \text{[InvNav]}$$

$$\frac{\begin{array}{l} p = (\mathcal{E}, x) \quad \theta' = \bot \\ \Sigma, \Delta, \theta' \vdash x \Downarrow v \end{array}}{\Sigma, \Delta \vdash p \Downarrow v} \quad \text{[Program]}$$

Fig. 17. Evaluation rules (Big Step SOS). "{| |}" is bag notation [5]

the value of the left expression, if it has at least one value, otherwise the value of the right expression. CONCAT combines all values, regardless of how many there are. Attributes can either be normal or have a derivation expression. For normal attributes a lookup is done in the attribute map of each entity passed into the expression (ATTR). The lookup of unset attributes fails, but these are filtered out. Derivations behave like a method call without arguments (AT2). Navigation works differently for navigating through a role or through an inverse. Navigating by role does a simple map lookup for each value (ROLENAV). Navigating by inverse does a reverse map lookup on the role it is the inverse of (INVNAV). Finally the program executes the main expression with the stores.

7 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 differs in the fact that our modeling language does not support imperative code, multiplicities are part of the type system and we implement relations of all degrees.

Classages is a language that also features relations [12]. 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 [4,22]. 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 [15]. 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 this approach does not

supply additional semantics for relations and thus cannot provide additional static type checking.

Languages with non first-class relations. In 1987 Rumbaugh was the first to add relations to a language [18]. His approach is pre-processor 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 [20]. Refer to Section 3 for details.

Queries of relations in object-oriented languages. The Java Query Language (JQL) adds queries to Java [21]. There is no additional support for relations, so navigation uses value-based joins like in SQL. LINQ also uses value-based joins [13]. 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 [19]. 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.

8 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.

Future work. We would like to add more aspects orthogonally to the type system. Our first candidates are ordered/unordered and unique/duplicates. It is worth exploring how well different aspects can be modelled orthogonally in a type system.

Also we would like to extend our language to provide type-and-multiplicity-safe operations on data. Adding or removing entities and relations might invalidate the multiplicity constraints on relations. We would like to catch these potential errors by static analysis and indicate to the programmer that he should catch that situation. The goal is to make sure that multiplicity-safe operations will never trigger runtime errors because a multiplicity constraint for a relation is violated. We would like to explore if we can ensure correct multiplicities at runtime statically.

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.T.: A relational model of object collaborations and its use in reasoning about relationships. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 323–346. Springer, Heidelberg (2007)
4. Bierman, G., Wren, A.: First-class relationships in an object-oriented language. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 262–286. Springer, Heidelberg (2005)
5. Buneman, P., Libkin, L., Suciu, D., Tannen, V., Wong, L.: Comprehension syntax. SIGMOD 23(1), 87–96 (1994)
6. Chen, P.P.: The entity-relationship model - toward a unified view of data. Tods 1(1), 9–36 (1976)
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. 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)
10. 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)
11. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: OOPSLA, pp. 444–463 (2010)
12. Liu, Y.D., Smith, S.F.: Interaction-based programming with classages. In: OOPSLA. pp. 191–209 (2005)
13. Meijer, E., Beckman, B., Bierman, G.M.: Linq: reconciling object, relations and xml in the .net framework. In: Sigmod, p. 706 (2006)
14. Mosses, P.D., New, M.J.: Implicit propagation in structural operational semantics. ENTCS 229(4), 49–66 (2009)
15. Stephen, Nelson, J.N., Pearce, D.J.: Implementing first-class relationships in java. Proceedings of RAOOL 8 (2008)
16. Noble, J.: Basic relationship patterns. Pattern Languages of Program Design 4 (1997)
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. Steimann, F.: Content over container: object-oriented programming with multiplicities. In: OOPSLA, pp. 173–186 (2013)
20. Visser, E.: WebDSL: A case study in domain-specific language engineering. In: GTTSE, pp. 291–373 (2007)
21. Willis, D., Pearce, D.J., Boyland, J.: Efficient object querying for java. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 28–49. Springer, Heidelberg (2006)
22. Wren, A.: Relationships for object-oriented programming languages. University of Cambridge, Computer Laboratory, Technical Report 702(UCAM-CL-TR-702) (November 2007)