A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships*

Stephanie Balzer[1], Thomas R. Gross[1], and Patrick Eugster[2]

[1] Department of Computer Science, ETH Zurich
[2] Department of Computer Science, Purdue University

Abstract. Understanding the *collaborations* that arise between the instances of classes in object-oriented programs is important for the analysis, optimization, or modification of the program. Relationships have been proposed as a programming language construct to enable an explicit representation of these collaborations. This paper introduces a *relational model* that allows the specification of systems composed of classes and relationships. These specifications rely in particular on *member interposition* (facilitates the specification of relationship-dependent members of classes) and on *relationship invariants* (facilitate the specification of the consistency constraints imposed on object collaborations). The notion of a *mathematical relation* is the basis for the model. Employing relations as an abstraction of relationships, the specification of a system can be formalized using discrete mathematics. The relational model allows thus not only the specification of object collaborations but also provides a foundation to reason about these collaborations in a rigorous fashion.

1 Introduction

The *collaborations* between objects are the key to understanding large object-oriented programs. Software systems do not accomplish their tasks with a single object in isolation, but only by employing a collection of objects — most likely instances of different classes — that exchange messages [1]. Unfortunately, class-based object-oriented programming languages do not provide sufficient means to explicitly specify these collaborations. Today's languages allow the description of objects through the programming language abstraction of a class, yet they lack a peer abstraction for object collaborations. Programmers must resort to the use of *references* to indicate collaborations and thereby often hide the intent and, at the same time, further complicate any analysis of a program since references are a powerful, all encompassing programming construct.

Conceptual modeling languages, such as the Unified Modeling Language (UML) [2] and the Entity-Relationship (ER) model [3], allow explicit representation of object collaborations through associations and relationships, respectively.

* This work was partially supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

E. Ernst (Ed.): ECOOP 2007, LNCS 4609, pp. 323–346, 2007.

The benefits of explicit representation of object collaborations also at the level of the programming language have been gaining increasing acceptance [4,5,6,7,8]. Languages devised in this spirit provide, in addition to classes, the programming language abstraction of a *relationship*. As classes allow the description of a collection of individual objects, relationships allow the description of a collection of groups of interacting objects. In both cases, the description involves the declaration of attributes and methods. Relationships furthermore indicate the classes of which the interacting objects are instances to delimit the scope of the collaboration.

The benefits of explicit representation of object collaborations through relationships at the level of the programming language are diverse [4,5,6,7,8]. Relationship-based implementations allow a *declarative* description of object collaborations. Class-based object-oriented implementations employ an *imperative* style since they represent object collaborations through references. The scope of a collaboration, for example, is explicitly declared in a relationship-based implementation. In a class-based object-oriented implementation, programmers must analyze the reference structure of the program to deduce the scope of the collaboration. Relationships are furthermore intrinsically bilateral, as both collaborators are known to the relationship. A class-based object-oriented implementation must deliberately introduce this bilateralism by providing a reference at each site of the collaboration. Relationship-based languages also allow the declaration of multiplicities (consistency constraints). In class-based object-oriented implementations, such multiplicities must be hand-coded by implementing the appropriate checks to enforce the constraints. These checks are most likely distributed among the classes participating in the collaboration, and this distribution carries the risk of introducing inconsistencies when the classes are updated. Relationships furthermore support the declaration of collaboration members. In class-based object-oriented implementations, such members must be taken care of manually. It appears overall that, as nicely put by Rumbaugh [5], "class-based object-oriented implementations of object collaborations *hide the semantic information* of collaborations but *expose their implementation details*" (emphasis added).

Unfortunately, the concepts supported by current relationship-based languages are not sufficient to specify object collaborations satisfactorily. Based on the example of an information system of a university (variations of this example can be found in several related publications [4,7,8,9]), we show which requirements of the system cannot be accommodated. Figure 1 shows the complete list of requirements for the university information system. Figure 2 depicts the corresponding UML class diagram of the system, and Fig. 3 sketches its implementation in a relationship-based language. There are several requirements that call for concepts not supported by current relationship-based languages. For example, the constraints that faculty members can neither substitute themselves nor each other (R7) and that students cannot assist courses they attend (R4) cannot be expressed through multiplicities. Also the restriction of the possible values attributes may assume, such as the year of study (R2), cannot be specified declaratively. The existence of entity properties that only apply when the

R1 The entities of the system are *students*, *courses*, and *faculty members*.
R2 For every student the *name*, a unique registration *number*, and the current *year* of study must be retained. The year of study cannot exceed 10. Courses must indicate their *titles*. Faculty members must list their *names*.
R3 Enrolled students must *attend* courses. When attending a course students can get a *mark* between 1 and 6.
R4 Students can *assist* courses as teaching assistants. Students cannot assist courses they are attending themselves. For every teaching assistant the *language of instruction* must be recorded. For every assisted course a maximal *group size* can be defined, which restricts the number of students that are assisted by a single teaching assistant. In case a maximal group size is prescribed for a course, then it must be guaranteed that the number of students assisted by a single teaching assistant must not exceed the maximal group size defined for that course.
R5 Students can *work for* a faculty member as research assistants, provided that they are at least in their third year. For every research assistant the *grant amount* paid can be retained.
R6 Every course must be *taught* by at least one faculty member.
R7 Every faculty member must name at least one other faculty member as *substitute*. No faculty member can be its own substitute, and two distinct faculty members cannot substitute each other.

Fig. 1. Requirements for the information system of a university

entity fulfills a particular role, such as the language of instruction for students assisting courses (R4), is a further example of an issue that can only be dealt with in current relationship-based languages by resorting to the introduction of auxiliary classes and further levels of indirection.

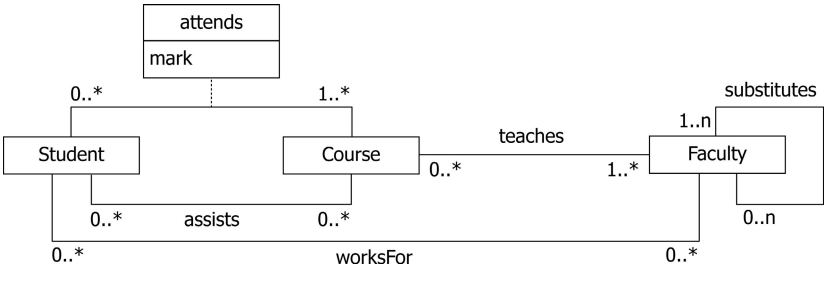

Fig. 2. UML class diagram modeling the simplified version of the information system of a university. The diagram uses an association class to allow the association `attends` to declare its own members.

In this paper, we show how to *specify object collaborations* in an *explicit* and *declarative* way. Our presentation is based on relationships but extends current relationship-based languages with concepts appropriate to accommodate the typical kinds of requirements imposed on software systems. In particular, we introduce *member interposition*, a concept allowing the specification of

```
relationship
  Attends (0-* Student learner, 1-* Course lecture) {
    int mark;
}
relationship
  Assists (0-* Student ta, 0-* Course course) {}

relationship
  WorksFor (0-* Student ra, 0-* Faculty supervisor) {}

relationship
  Teaches (1-* Faculty lecturer, 0-* Course lecture) {}

relationship
 Substitutes (1-* Faculty substitute, 0-* Faculty substituted){}
```

Fig. 3. Implementation of the running example in a language supporting relationships. The code combines features present in RelJ [4] and/or the Data Structure Manager (DSM) [5]. Details not relevant to the discussion have been omitted.

relationship-dependent members of classes, and *relationship invariants*, a concept allowing the specification of the consistency constraints of relationships. Since we use *mathematical relations* as the fundamental abstractions to reason about relationships, we can express relationship invariants by means of the mathematical properties of the relations underlying the relationships. The abstraction of a relation furthermore allows a formalization of the concepts we introduce relying entirely on discrete mathematics.

The remainder of the paper is organized as follows: Sect. 2 introduces relations, the abstractions underlying the relational model. Sects. 3 and 4 detail member interposition and relationship invariants, respectively. Sect. 5 discusses further issues related to the presented concepts. Sect. 6 provides design guidelines for a programming language accommodating specifications as presented in this paper. Sect. 7 lists the related work and Sect. 8 concludes the paper.

2 Relations

In this section we introduce relations, the driving forces underlying the concepts presented in this paper. We also set up our terminology.

2.1 Abstracting Object Collaborations

The existence of an appropriate abstraction to reason about systems composed of classes and relationships is a prerequisite to their specification. We use the notion of a *mathematical relation* as an abstraction of a relationship. Figure 4 depicts the relationships `Attends` and `Teaches`. As classes describe the common properties of a collection of individual objects, we abstract them as *sets of objects*.

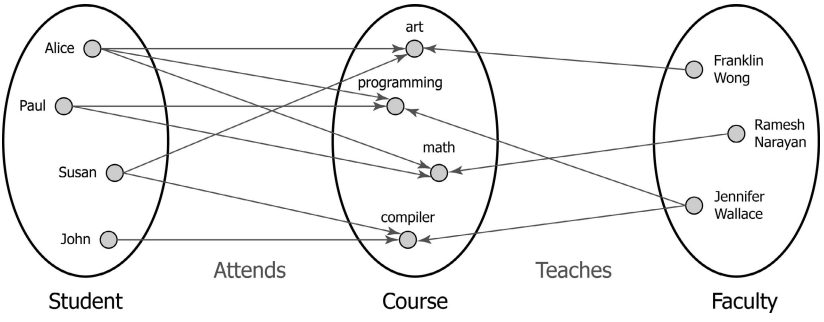

Fig. 4. Graphical representation of the `Attends` and `Teaches` relationship: classes (ellipses) are sets of objects (circles) and relationships (arrows in between ellipses) are sets of object tuples

As relationships describe the common properties of a collection of groups of collaborating objects, we abstract them as *sets of object tuples* and consequently as *relations*. Figure 4 thus contains the sets

$$\begin{aligned} \mathit{Student} &= \{\ \mathit{Alice},\ \mathit{Paul},\ \mathit{Susan},\ \mathit{John}\} \\ \mathit{Course} &= \{\ \mathit{art},\ \mathit{programming},\ \mathit{math},\ \mathit{compiler}\} \\ \mathit{Faculty} &= \{\ \mathit{Franklin\ Wong},\ \mathit{Ramesh\ Narayan},\ \mathit{Jennifer\ Wallace}\} \end{aligned}$$

and the relations

$$\begin{aligned} \mathit{Attends} = \{\ & \mathit{Paul} \mapsto \mathit{programming},\ \mathit{Paul} \mapsto \mathit{math},\ \mathit{John} \mapsto \mathit{compiler}, \\ & \mathit{Alice} \mapsto \mathit{art},\ \mathit{Alice} \mapsto \mathit{programming},\ \mathit{Alice} \mapsto \mathit{math}, \\ & \mathit{Susan} \mapsto \mathit{art},\ \mathit{Susan} \mapsto \mathit{compiler}\} \\ \mathit{Teaches} = \{\ & \mathit{Jennifer\ Wallace} \mapsto \mathit{programming},\ \mathit{Jennifer\ Wallace} \mapsto \mathit{compiler}, \\ & \mathit{Ramesh\ Narayan} \mapsto \mathit{math},\ \mathit{Franklin\ Wong} \mapsto \mathit{art}\} \end{aligned}$$

Thanks to mathematical relations we can model a system composed of classes and relationships using discrete mathematics. The resulting *relational model* of a system then allows us to reason about a system composed of classes and relationships in a rigorous fashion. From its model, we can derive the specification of a system. The university information system yields the following initial model:

$$\begin{aligned} \mathit{Attends} &\subseteq \mathit{Student} \times \mathit{Course} \\ \mathit{Assists} &\subseteq \mathit{Student} \times \mathit{Course} \\ \mathit{WorksFor} &\subseteq \mathit{Student} \times \mathit{Faculty} \\ \mathit{Teaches} &\subseteq \mathit{Faculty} \times \mathit{Course} \\ \mathit{Substitutes} &\subseteq \mathit{Faculty} \times \mathit{Faculty} \end{aligned}$$

2.2 Terminology and Restrictions

Before continuing the presentation of the specification of object collaborations based on relations, we briefly set up the terminology used in this paper. We

restrict the specification of object collaborations to the non-concurrent case and — because of space constraints — we do not discuss inheritance either.

Class: We consider classes as *types* and also as *sets*. Such a set contains objects that are instances of the type defined by the class declaration.

Relationship: We consider a relationship to be both a *type* and a *relation*. The relation contains the *object tuples* that are instances of the type defined by the relationship declaration.

Participants: The participants of a relationship are the *carrier sets* (i.e., classes) of the relation defining the relationship.

Roles: The participants of a relationship declaration can be named to indicate the conceptual *role* the particular class plays in the relationship.

3 Member Interposition

Some properties of objects only apply when the object is fulfilling a particular role [10]. The attributes `instructionLanguage` (see Fig. 1, R4) and `grantAmount` (R5) are examples of such properties since these properties are required only for teaching and research assistants, respectively, but not for all students. Thus, the selection of properties that are required for an object depends on the relationship(s) the object takes part in.

3.1 Problem Description and Solution

Member interposition accommodates relationship-dependent properties of objects. Member interposition allows us to define properties as part of the role a particular class fulfills in a relationship. Figure 5 gives an example. Both the attributes `instructionLanguage` and `grantAmount` are declared in the relationships on which these attributes depend and as part of the roles played by the classes to which these attributes apply (are interposed into). Attribute `instructionLanguage` is declared in relationship `Assists` for the role teaching assistant (`ta`), attribute `grantAmount` in relationship `WorksFor` for the role research assistant (`ra`).

Without member interposition, we would have to use the role object [11] and extension object [12] design patterns, respectively. We then would subclass `Student` to provide specializations both for teaching and research assistants and would need to introduce an additional level of indirection to represent the possible roles students can play and to allow dynamic casts between these roles. Member interposition, on the other hand, allows us to accommodate relationship-dependent properties of classes without resorting to inheritance and role classes.

Relationships can declare both interposed members and non-interposed members. Attribute `mark` in relationship `Attends` is an instance of a member that is not interposed. Whereas an interposed member describes a class that plays a particular role in a relationship, a non-interposed member describes the

```
relationship Attends
  participants (Student learner, Course lecture) {
  int mark;
}
relationship Assists
  participants (Student ta, Course course) {
    // attribute interposed into role ta
  String >ta instructionLanguage;
}
relationship WorksFor
  participants (Student ra, Faculty supervisor) {
    // attribute interposed into role ra
  int >ra grantAmount;
}
```

Fig. 5. Relationship members are declared either at the level of the relationship or, through *member interposition*, at the level of a participating role. Interposed members are declared using the ">" symbol and are depicted underlined.

collaboration that exists between the participants of a relationship. We therefore also refer to interposed members as *participant-level* members and to non-interposed members as *relationship-level* members. Intuitively (see Fig. 4), we can imagine an interposed member as being attached to each object (circle) of the class (ellipse) that is the target of interposition. A non-interposed member, on the other hand, is attached to each object tuple (arrow) of the relationship. In the current specification, the attribute `instructionLanguage` records per teaching assistant the language of instruction. If we interposed attribute `instructionLanguage` into the role `course` instead of the role `ta`, we could indicate in what language a particular course must be assisted. A third option would be to declare attribute `instructionLanguage` as a non-interposed member. In this case, teaching assistants would be allowed to use different languages for different courses.

Like non-interposed members, interposed members are part of the interface[1] of their defining relationships (and not part of the interface of the classes they are interposed into). This treatment has two consequences. First, the names of interposed members must be unique only within their defining relationship. An interposed member can therefore be named the same as a member of the class that is the target of interposition or the same as an interposed member of a different relationship that has the same target of interposition. In both cases, separate copies of these members are maintained. Second, operations to access interposed members must be called on the relationship and are not allowed to be called directly on the targeted class. According to Snyder [14], encapsulation in class-based object-oriented programming languages aims to minimize the module interdependences through the application of strict external interfaces. Since

[1] We use the term interface as introduced by Parnas [13].

member interposition leaves the interfaces of the classes being the target of interposition unchanged, the encapsulation of these classes remains unaffected.

3.2 Formalization

Using the abstraction of a relation and the means provided by discrete mathematics we can formalize the interposed and non-interposed attributes of Fig. 5 as follows:

$$Attends\_mark \in Attends \nrightarrow [1 \,..\, 6] \tag{3.1}$$

$$Assists\_instructionLanguage \in \mathrm{dom}(Assists) \rightarrow String \tag{3.2}$$

$$WorksFor\_grantAmount \in \mathrm{dom}(WorksFor) \nrightarrow \mathbb{N} \tag{3.3}$$

As illustrated by (3.1), we can model a relationship-level attribute as a relation from a relation to the set of possible values the attribute may assume (see Table 1 for an explanation of the notation used). In the example, the relationship-level attribute `mark` is a partial function from the relation *Attends* to the set of integer numbers ranging from 1 to 6. Note that we restrict the range of the function to $[1 \,..\, 6]$ as imposed by R3 in Fig. 1. Participant-level attributes (3.2) and (3.3), on the other hand, are relations from the domain or range of a relation to the set of possible values the attribute may assume. The interposed attribute `instructionLanguage` (3.2), for example, is a relation that has the domain of the relation *Assists* — which is a subset of the set *Student* — as its domain, and a set of strings as its range. Note that we use a total function for the relation since we need to retain the language of instruction for every student assisting a course (R4).

Table 1. Mathematical notation as defined in [15]

Symbol	Description
$\mapsto$	Pair constructing operator
$S \leftrightarrow T$	Set of binary relations from S to T
$S \twoheadleftarrow\!\!\!\!\!\rightarrow T$	Set of surjective relations from S to T
$S \leftrightarrow\!\!\!\!\!\rightarrow T$	Set of total relations from S to T
$S \nrightarrow T$	Set of partial functions from S to T
$S \rightarrow T$	Set of total functions from S to T
$S\,;T$	Forward composition of relations S and T
S^{-1}	Inverse of relation S
$\mathrm{dom}(S)$	Domain of relation S
$\mathrm{ran}(S)$	Range of relation S
$S[m]$	Image of the set m under the relation S
$\mathrm{card}(m)$	Number of elements of set m
$\mathrm{id}(m)$	Identity relation built on set m

Since member interposition targets at the role of a participant rather than at the class as a whole, it is possible to selectively add properties to objects that are instances of the same class, but play different roles in the same relationship (relationship `Substitutes`, for example). In such a case, we formalize the relation defining the relationship as a relation from one subset of the participant to another subset of the participant, with each subset containing the objects that play a particular role in the relationship.

4 Relationship Invariants

Current relationship-based languages do not provide the appropriate means to declare consistency constraints other than multiplicities. As demonstrated by the running example of this paper, the existence of more elaborate constraints, such as the restriction that students cannot assist courses they are attending (Fig. 1, R4), are an important trait of object collaborations. We introduce the concept of *relationship invariants* to express consistency constraints required for the specification of object collaborations.

Invariants proved viable for the specification of consistency constraints in a number of class-based object-oriented programming and specification languages, such as the Eiffel programming language [16,17], the Spec# programming system [18], and the behavioral interface specification language for Java, JML (Java Modeling Language), and its verification tools [19,20]. Whereas invariants of class-based object-oriented programming languages are imposed on individual objects (object invariants) or on the class as a whole (static class invariants) [21], we allow invariants to range over several classes by imposing them on one or several relationships. As we maintain a set-oriented view of classes and relationships, invariants implicitly quantify over the objects or object tuples contained in the set the invariants are imposed on. Classical invariants of class-based object-oriented programming and specification languages are different: such invariants are restricted to individual objects and classes, respectively. The restricted scope of classical invariants makes the verification of invariants particularly challenging in case an invariant involves references [22,23].

We distinguish between *intra-relationship* and *inter-relationship* invariants, and between *value-based* and *structural* invariants. The first category denotes the scope of the invariant. An intra-relationship invariant is imposed on a single relationship and thus restricts the collaboration of the participants within that relationship. An inter-relationship invariant involves several relationships and thus defines how relationships relate to each other. The second category distinguishes whether values that relationships or participating classes assume for their members are taken into account for the invariant specification. A value-based invariant defines the values or the range of values the elements in the scope of the invariant declaration are allowed to assume for the specified members. A structural invariant restricts the possible ways different elements in the scope of the invariant declaration can be paired up irrespective of the values these elements assume for their members. The two categories are orthogonal, yielding

four kinds of invariants. We provide a formalization of each kind of invariant using the abstraction of a relation for a relationship.

4.1 Structural Invariants

We start the presentation of the different kinds of invariants with structural invariants as they are similar to multiplicity restrictions. In fact, multiplicity restrictions are a subset of structural intra-relationship invariants.

Structural Intra-Relationship Invariants. The requirements document of the university information system (see Fig. 1) lists several structural invariants, such as the restrictions that enrolled students must attend courses (R3) and that every faculty member must name at least one other faculty member as substitute (R7). These restrictions, expressed by *multiplicities* in current relationship-based programming languages, define the structural characteristics of a relationship and can thus be formalized by indicating the structural properties of the relations defining the relationship. For example, the `(0..*,1..*)` multiplicity of the relationship `Attends` can be formalized as a total relation, and the `(1..*,0..*)` multiplicity of the relationship `Substitutes` can be formalized as a surjective relation, as outlined by (4.1) and (4.5), respectively, in the following:

$$\mathit{Attends} \in \mathit{Student} \nleftrightarrow \mathit{Course} \tag{4.1}$$

$$\mathit{Assists} \in \mathit{Student} \leftrightarrow \mathit{Course} \tag{4.2}$$

$$\mathit{WorksFor} \in \mathit{Student} \leftrightarrow \mathit{Faculty} \tag{4.3}$$

$$\mathit{Teaches} \in \mathit{Faculty} \nleftrightarrow\!\!\!\!\rightarrow \mathit{Course} \tag{4.4}$$

$$\mathit{Substitutes} \in \mathit{Faculty} \nleftrightarrow\!\!\!\!\rightarrow \mathit{Faculty} \tag{4.5}$$

There are additional structural invariants present in the running example, for example, that no faculty member can be his or her own substitute and that two distinct faculty members cannot substitute each other (R7). These structural constraints, not expressible through multiplicities, define the asymmetry (4.6) and irreflexiveness (4.7) of the `Substitutes` relationship and can be formalized as follows:

$$\mathit{Substitutes} \cap \mathit{Substitutes}^{-1} = \varnothing \tag{4.6}$$

$$\mathit{Substitutes} \cap \mathrm{id}(\mathit{Faculty}) = \varnothing \tag{4.7}$$

Based on the example of the `Substitutes` relationship, Fig. 6 illustrates how structural intra-relationship invariants can be specified as part of relationship declarations.

Structural Inter-Relationship Invariants. According to the requirements document of the university information system (see Fig. 1) students are not allowed to assist courses they are attending themselves (R4). This requirement also represents a structural invariant, but, in contrast to the invariants discussed in the previous section, this invariant encompasses several relationships:

```
relationship Substitutes
  participants (Faculty substitute, Faculty substituted) {

  invariant
    surjectiveRelation(Substitutes) &&
    asymmetric(Substitutes) &&
    irreflexive(Substitutes);
}
```

Fig. 6. Relationship `Substitutes` with a structural intra-relationship invariant: the relation defining the relationship is surjective, asymmetric, and irreflexive

the relationship `Assists` ("students are not allowed to *assist* courses...") and the relationship `Attends` ("...they are *attending* themselves"). To satisfy the requirement, the two defining relations of the relationships must be disjoint:

$$Attends \cap Assists = \varnothing \tag{4.8}$$

Figure 7 illustrates how the structural inter-relationship invariant (4.8) can be specified as part of a program composed of classes and relationships. Unlike an intra-relationship invariant, which can be directly listed as part of the relationship declaration, an inter-relationship declaration appears outside of the scope of the relationship declarations it is imposed on.

```
invariant (Attends, Assists) attendsAssistsDisjointness:
  Attends intersection Assists == emptySet;
```

Fig. 7. Structural inter-relationship invariant guaranteeing that teaching assistants cannot attend the courses they are assisting. An inter-relationship invariant can be named and indicates the relationships it is imposed on in parentheses.

4.2 Value-Based Invariants

Value-based intra- and inter-relationship invariants bear resemblance to traditional invariants of class-based object-oriented programming languages as traditional invariants are assertions on the values the fields of an object or a class may assume. Value-based invariants, however, exceed the scope of traditional invariants as they range over several classes and relationships.

Value-Based Intra-Relationship Invariants. The requirements document of the university information system (see Fig. 1) demands that students must be at least in their third year to become research assistants (R5). This requirement can be expressed through a value-based intra-relationship invariant. As demonstrated by (4.10) below we can formalize the invariant by requiring that

```
relationship WorksFor
  participants (Student ra, Faculty supervisor) {
    // attribute interposed into role ra
  int >ra grantAmount;

  invariant
    relation(WorksFor) &&
    ra.year > 2 &&
    partialFunction(grantAmount) in N;
}
```

Fig. 8. Relationship invariant consisting of a structural intra-relationships invariant `relation(WorksFor)` and two value-based intra-relationship invariants guaranteeing that research assistants are at least in their third year of study (`ra.year > 2`) and that the amount of funding the student receives is optional and a natural number (`partialFunction(grantAmount) in N`)

the range of the forward composition $WorksFor^{-1}$; *Student_year* is a subset of the set of integer numbers ranging from 3 to 10. The forward composition yields the set of pairs of faculty members and integer numbers, with one pair for each group of research assistants that are supervised by the same faculty member and that share the same year of study. The relation *Student_year* (4.9) abstracts the attribute `year` of class `Student`.

$$Student\_year \in Student \rightarrow [1\,..\,10] \tag{4.9}$$

$$\mathrm{ran}(\,WorksFor^{-1}\,;\,Student\_year) \subseteq [3\,..\,10] \tag{4.10}$$

Interestingly, the invariant (4.10) involves a member of a participant and not a member of a relationship. As the constraint imposed on the member depends on the relationship — the year of study needs to be considered only for research assistants but not for students in general — it cannot be declared as a class invariant (see Sect. 5.2) but must be declared as a relationship invariant. In a mere class-based implementation of the running example with support for traditional invariants, the definition of the constraint would have to account for this dependence. To guard the evaluation of the invariant, a resulting object invariant would most likely introduce an implication of the form `supervisor != null ==> this.year > 2`.

Figure 8 illustrates how the value-based intra-relationship invariant (4.10) can be specified as part of the declaration of relationship `WorksFor`. The figure furthermore reveals that an invariant declaration can consist of several kinds of invariants. Besides the value-based intra-relationship invariant imposing the constraint just discussed, Fig. 8 lists the structural intra-relationship invariant `relation(WorksFor)` and a further value-based intra-relationship invariant `partialFunction(grantAmount) in N` defining the nature of the interposed member `grantAmount`. The corresponding formalization of the attribute `grantAmount` was introduced in Sect. 3.2.

```
invariant (Attends, Assists) enoughAssistants:
  forAll c (isDefined(Assists.select(course==c).maxGroupSize)
    ==> numberOf(Attends.lecture.select(c)) <=
      numberOf(Assists.course.select(c)) *
      Assists.select(course==c).maxGroupSize);
```

Fig. 9. Value-based inter-relationship invariant guaranteeing that there are enough teaching assistants per course. The use of role names as in `Attends.lecture` allows the retrieval of the set of objects participating in the relationship and playing the indicated role. The **`select`** operator allows the retrieval of the set of objects (when applied to a role) or set of object tuples (when applied to the relationship) that match the condition indicated in parentheses. For further details see Sect. 6.

Value-Based Inter-Relationship Invariants. The requirements document of the university information system (see Fig. 1, R4) prescribes that the number of students assisted by a single teaching assistant for a particular course does not exceed the maximal group size defined for that course, if defined at all. This requirement guarantees that enough teaching assistants are recruited for a particular course. We can formalize this restriction by requiring that, for every course, the number of students attending the course ($\mathrm{card}(\mathit{Attends}^{-1}[\{c\}])$) is less than or equal to the number of assistants assisting the course ($\mathrm{card}(\mathit{Assists}^{-1}[\{c\}])$) multiplied by the maximal group size for the course ($\mathit{Assists\_maxGroupSize}(c)$). As indicated by the implication in (4.11), the inequality is evaluated for a course only that is currently assisted and for which the attribute `maxGroupSize` is defined.

$$\begin{aligned} \forall c \cdot (c \in \mathrm{dom}(\mathit{Assists\_maxGroupSize}) \quad &\Rightarrow \\ \mathrm{card}(\mathit{Attends}^{-1}[\{c\}]) \quad &\leq \\ \mathrm{card}(\mathit{Assists}^{-1}[\{c\}]) * \mathit{Assists\_maxGroupSize}(c)) & \end{aligned} \tag{4.11}$$

Figure 9 shows the corresponding program specification of the value-based inter-relationship invariant (4.11). In the example, we need to introduce explicit quantification as the invariant must hold only for selected constituent objects of the tuples involved.

5 Discussion

The use of relationships together with the concepts introduced in this paper influences not only the specification of object collaborations but also the development of programs composed of classes and relationships in general. In this section, we discuss some consequences.

5.1 References

The introduction of relationships changes the purpose of references. In class-based object-oriented programs references allow the implementation of object

collaborations. For example, students keep references to the list of courses they attend. With explicit relationships, on the other hand, classes no longer need to maintain references to (instances of) the classes they collaborate with as the description of this collaboration is "out-sourced" to the corresponding relationship.

Relationships, however, need a kind of reference to access the objects that participate in a relationship. It is questionable, though, whether traditional references are the appropriate means to implement the "awareness" of a relationship of its participants. To answer this question, we must consider what the characteristics of references are and which traits of these characteristics are required in the case of relationships. References can be used in two different ways: (*i*) to access the *artifact* that the reference refers to and (*ii*) to read or change the *value* (object identifier in a class-based object-oriented context) of the reference. With respect to relationships, the first use of references is clearly desired — it must be possible to access the objects that participate in a relationship. However, an object tuple should not be allowed to change its identity by replacing (or possibly erasing) any of its constituent objects. Relationships therefore need *restricted forms* of references that allow access of the constituent objects but prohibit direct manipulation of the values assigned to references. Role names, for example, can serve that purpose.

Of course, it must be possible to change the participation of objects in relationships. Because we consider classes and relationships as sets — sets of objects and sets of object tuples, respectively — changes in relationship participation break down to adding and removing object tuples to and from relationships. These operations encompass the relationship as a whole and must therefore be executed outside of the scope of the targeted relationship (see Sect. 6.1).

5.2 Class Invariants

A specification of programs composed of classes and relationships must include the declaration of class invariants besides the declaration of relationship invariants. *Class invariants* allow the specification of the consistency constraints that are imposed on the instances of individual classes. Since classes do not describe their collaborations with other classes, class invariants have an *intra-class* scope and are purely *value-based*. To restrict the possible values objects can assume for their members, we abstract object members as relations from classes (sets) to the sets of possible values their members may assume. The mathematical properties of these relations then express the class invariant.

Equations (5.1), (5.2), and (5.3) show the relations abstracting the members of class `Student`. Equation (5.2) uses a total injection from the set *Student* to the set of natural numbers to express that every student must have a number which is unique.

$$Student\_name \in Student \to String \tag{5.1}$$

$$Student\_number \in Student \rightarrowtail \mathbb{N} \tag{5.2}$$

$$Student\_year \in Student \to [1 \,..\, 10] \tag{5.3}$$

```
class Student {
  String name;
  int number;
  int year;

  invariant
    totalFunction(name) &&
    totalInjection(number) in N &&
    totalFunction(year) in [1..10];
}
```

Fig. 10. Specification of the class invariant of `Student` restricting the possible values class members can assume

Figure 10 shows the corresponding declaration of the class invariants of class `Student`. The invariant that specifies that student numbers must be unique highlights the benefits of treating classes and class members as sets and relations, respectively. As opposed to its counterpart in a class-based implementation, it is simple and clear-cut. In a class-based setting, on the other hand, the specification of the same constraint would demand a more extensive invariant. To express the injectivity of the relation *Student_number*, a static class invariant would need to be declared which uses explicit quantification to range over all instances of the class and to make sure that the attribute `number` is different for every instance.

5.3 Invariant Preservation

The use of invariants as part of the declaration of classes and relationships raises the question of their verification. Irrespective of the approach taken — run-time verification (*dynamic*) or compile-time verification (*static*) — the invariants imposed on a system composed of classes and relationships must be preserved from one state to the other along state transitions of the system.

The relational model can help to substantially decrease the number of transitions that must be inspected to verify the invariant. Thanks to the categorization of invariants we can identify for each kind of invariant the operations that cause state transitions that potentially endanger the invariant. For structural invariants, for example, the operations causing such transitions are the addition to and removal of objects from classes, and the addition to and removal of object tuples from relationships. If we consider, in addition to its category, also the mathematical properties of an invariant, we can delimit the cases in which state transitions occur that potentially endanger the invariant.

In the most general case of a relation as a structural intra-relationship invariant the following interdependence between the relationship and its participants exists:

$$R \in A \leftrightarrow B \Leftrightarrow \forall a, b \cdot (a \mapsto b \in R \Rightarrow a \in A \wedge b \in B) \tag{5.4}$$

From (5.4) we can delimit the following invariant-endangering operations:

- the removal of an object from a class if the object participates in a relationship with the class being a participant of that relationship
- the addition of an object tuple if the constituent objects are not part of the participants of the relation.

The number of invariant-endangering operations increases with the restrictiveness of the relation. In case of a total relation, for example, we can delimit the following invariant-endangering operations:

- the addition of an object to a class that is the domain of the total relation
- the removal of an object from a class that is the domain of the total relation
- the addition of an object tuple if the constituent objects are not part of the participants of the total relation
- the removal of an object tuple from the total relation if no other object tuple exists in that relation that contains the first constituent object of the tuple to be removed.

The handling of these invariant-endangering operations must be left to the respective programming language or system that implements the specification concepts introduced in this paper. An implementation could, for example, deal with certain invariant-endangering operations by executing a corresponding corrective action to maintain the invariant. A further implementation concern is to determine the granularity of atomic sequences of operations. Most likely, an implementation will provide the means to combine several invariant-endangering operations in one atomic unit and thus allow a further decrease of the verification load.

We expect the relational model of object collaborations to be helpful with verifying invariants statically. Thanks to its foundation in discrete mathematics, a relation model describing a concrete system composed of classes and relationships could easily be transformed to the input required by a theorem prover or model checker, which then would allow the verification of the system.

6 Language Design Issues

In this section we sketch the main features of a programming language that incorporates the specification concepts introduced in this paper.

6.1 Three Dimensions of Problem Decomposition

In their seminal paper on programming with abstract data types, Liskov et al. [24] introduce two forms of programming language abstractions: procedures (functional abstraction) and operation clusters (abstract data types). We regard the separation of functional decomposition from data decomposition to be valuable as it allows us to separate the definition of artifacts from their use. Due to our focus on the specification of object collaborations, we complement

the abstractions introduced by Liskov et al. with the abstraction representing object collaborations. A programming language that incorporates the specification concepts introduced in this paper thus needs to support the following language abstractions:

- *Class (data decomposition)*: Programming language abstraction representing classes as defined in Sect. 2.2.
- *Relationship (collaboration decomposition)*: Programming language abstraction representing relationships as defined in Sect. 2.2.
- *Application (functional decomposition)*: Programming language abstraction comprising a number of procedures to manipulate the sets of objects and object tuples contained in a program.

6.2 Language Definition

A programming language that incorporates the specification concepts introduced in this paper must support the types *ValueType*, *ClassName*, *RelationshipName*, *Object*⟨*ClassName*⟩, and *Query*⟨*Set*⟩. A *ValueType* is a type with a value type semantics. Whereas both *ClassName* and *RelationshipName* are types that denote sets, *Object*⟨*ClassName*⟩ is a parameterized type that stands for a particular instance of the class provided as an argument. A *Query*⟨*Set*⟩ is also a parameterized type that represents sets of objects or object tuples. Possible arguments to a query type are class names, relationship names, or any expressions composed of relationship names and relational operators yielding a set as a result. Both parameterized types are instances of *HandleType*, which represents a Java final-like reference to either an object or a set of objects and object tuples, respectively.

Like Bierman and Wren [4] we use tables and maps (see Fig. 11) to formalize the declarations appearing in a program devised in the language under discussion. We have tables for classes, relationships, inter-relationship invariants, and for applications. Each table is a map from a name (class name, for example) to a definition (class definition, for example). Definitions are tuples with the elements being sets or further maps. For example, a class definition is a tuple ($\mathcal{A}$,$\mathcal{M}$,*ci*) where $\mathcal{A}$ is a map from attribute names to attribute types, $\mathcal{M}$ is a map from method names to method definitions, and *ci* is the class invariant body. The signature definitions in Fig. 11 reveal an important characteristic of the programming language: both the attributes of classes and relationships are of value type only (see Sect. 5.1 for a further discussion). As relationships must have access to their participating objects, *RelMethodMap* lists the set *RoleName* in its range. Unlike classes and relationships, applications are neither types nor do they declare invariants. Applications are mere procedural modules that consist of a number of variables and procedures. As these procedures need to instantiate classes and need to add to and remove objects from classes and object tuples from relationships, respectively, applications can declare variables of type *HandleType*.

6.3 Creation, Addition, Removal, and Retrieval

An appropriate programming language must provide built-in operators to *instantiate* classes, to *add* to and *remove* objects from classes, and to add to and

$$
\begin{aligned}
\mathit{ClassTable} &\in \mathit{ClassName} \rightarrow \mathit{AttrMap} \times \mathit{ClMethodMap} \times \mathit{ClassInvBody} \\
\mathit{RelationshipTable} &\in \mathit{RelationshipName} \rightarrow \mathit{RoleMap} \times \mathit{AttrMap} \times \mathit{RelMethodMap} \\
&\quad \times \mathit{RelInvBody} \\
\mathit{InterRelInvTable} &\in (\mathit{InvName} \times (\mathit{RelationshipName} \times \cdots \times \mathit{RelationshipName})) \\
&\quad \rightarrow \mathit{InterRelInvBody} \\
\mathit{ApplicationTable} &\in \mathit{ApplicationName} \rightarrow \mathit{VarMap} \times \mathit{ProcedureMap} \\
\mathit{RoleMap} &\in \mathit{RoleName} \rightarrow \mathit{ClassName} \\
\mathit{AttrMap} &\in \mathit{AttrName} \rightarrow \mathit{ValueType} \\
\mathit{ClMethodMap} &\in \mathit{MethodName} \rightarrow \mathit{ArgMap} \times \mathit{LocalMap} \times \mathit{ValueType} \times \mathit{MethodBody} \\
\mathit{RelMethodMap} &\in \mathit{MethodName} \rightarrow \mathit{ArgMap} \times \mathit{LocalMap} \times \mathit{RoleName} \times \mathit{ValueType} \\
&\quad \times \mathit{MethodBody} \\
\mathit{ArgMap} &\in \mathit{ArgName} \rightarrow \mathit{ValueType} \\
\mathit{LocalMap} &\in \mathit{VarName} \rightarrow \mathit{ValueType} \\
\mathit{VarMap} &\in \mathit{VarName} \rightarrow (\mathit{ValueType} \cup \mathit{HandleType}) \\
\mathit{ProcedureMap} &\in \mathit{ProcName} \rightarrow \mathit{ProcArgMap} \times \mathit{ProcLocalMap} \\
&\quad \times (\mathit{ValueType} \cup \mathit{HandleType}) \times \mathit{ProcBody} \\
\mathit{ProcArgMap} &\in \mathit{ArgName} \rightarrow (\mathit{ValueType} \cup \mathit{HandleType}) \\
\mathit{ProcLocalMap} &\in \mathit{VarName} \rightarrow (\mathit{ValueType} \cup \mathit{HandleType})
\end{aligned}
$$

Fig. 11. Signatures of class, relationship, and application tables and associated maps

remove object tuples from relationships. As both classes and relationships are sets, the addition and removal operators are conceptually equivalent with set union and set subtraction.

Figure 12 shows a program fragment implementing the running example. The fragment consists of several class and relationship declarations and one application declaration. Class `Student` declares a constructor to allow creation of a single object. The constructor is called in procedure `initialize` in application `UniversityInformationSystem` and assigned to the object handle `alice`. The object denoted by `alice` is then added to class `Student` in line 27. For every course Alice is attending, a corresponding student-course pair is added to the relationship `Attends` (28-30). In the examples, the **`add`** operator is used to add single objects and single object tuples, respectively. The **`add`** operator, however, can also be used to add a set of objects and a set of object tuples to a class and relationship, respectively. The same explanations apply likewise to the **`remove`** operator.

An appropriate programming language must further provide built-in operators to *retrieve* sets of objects and sets of object tuples. These operators must enable the retrieval of the set of objects playing a particular role in a relationship and the retrieval of the set of objects or object tuples that satisfy a given condition.

Figure 12 illustrates the use of the **`select`** operator to retrieve the set of object tuples that match the condition provided as an argument. In line 21, all faculty-course pairs are retrieved with the object denoted by the handle

```
class Student {
 Object<Student> Student(String name, int number, int year) {
  this.name = name;
  this.number = number;
  this.year = year;
 }...
}...
relationship Attends
 participants (Student learner, Course lecture) {...}
...
application UniversityInformationSystem {
 // handles to objects
 Object<Student> alice, john, susan, paul;
 Object<Faculty> jenniferWallace, rameshNarayan, franklinWong;
 Object<Course> programming, math, compiler, art;
 ...
 void main() {
  initialize();
  // assign all students of Franklin Wong the grade 6
  Query<Teaches.lecture> coursesFw;
  coursesFw = Teaches.select(lecturer==franklinWong).lecture;
  Attends.select(lecture==coursesFw).setMark(6);
 }
 void initialize() {
  ...
  alice = new Student("Alice", 778, 1);
  Student.add(alice);
  Attends.add(alice, programming);
  Attends.add(alice, math);
  Attends.add(alice, art);
 }...
}
```

Fig. 12. Program fragment consisting of several class and relationship declarations and one application declaration. The fragment shows the creation of an object (26), the addition of an object to a class (27), and the addition of object pairs to a relationship (28-30). Line 20 illustrates the use of queries as handles to a set that contains the result of a retrieval operation. Line 22, furthermore highlights the set-oriented character of the language: method `setMark` is called on the set of student-course pairs such that the students contained in the set attend a class taught by Franklin Wong.

`franklinWong` playing the role of the `lecturer`. On this result set, the role operator `lecture` is applied to return only the set of courses that are taught by Franklin Wong. Of interest in the example is also the declaration of a *query type* in line 20. According to the argument of the type, the query `coursesFw` is a handle to a set of objects that are instances of type `Course`. Queries are a powerful construct as they allow us to type intermediate retrieval results. In the example, the query encompasses only one relationship, however, as a query accepts any argument that is a set, any expression composed of relationship

names and relational operators yielding a set as a result can be declared as an argument of a query.

As demonstrated by the instructions on line 20-22, relationships ease the handling of object collections. Willis et al. [25] introduced a prototype extension to Java that supports the concept of first-class queries to dispose of iterators otherwise needed to traverse object collections. However, such first-class queries still need to explicitly indicate the matching attributes that establish the "join" condition. With relationships, on the other hand, queries do not have to indicate the join conditions as they are implicitly established through the relationship declaration.

6.4 Atomic Procedures

Separating application declarations from class and relationship declarations allows us furthermore to separate the declaration of invariants from their verification. As the procedures of applications are the only ones to manipulate classes and relationships, they are also the only ones to endanger the invariants established on the classes and relationships. Therefore, in a programming language that incorporates the specification concepts introduced in this paper, the body of an application procedure defines the granularity of atomic sequences of operations. Invariants, consequently, must hold only on entry and exit of procedures, but not during procedure execution. In Fig. 12, for example, the structural intra-relationship invariant of relationship `Attends` is temporarily violated after the addition of the object denoted by `alice` to class `Student` in line 27 (relationship `Attends` is defined by a total relation). Only after the addition of the first student-course pair to the relationship `Attends` for a course Alice is attending, the invariant is re-established and preserved until the end of the procedure.

7 Related Work

In this section we discuss the related work. To be consistent with previous sections, we use our terminology for the discussion and indicate the actual terms used in the respective publication(s), if different, in parentheses.

Rumbaugh [5] first discovered the important role relationships (relations) play in object-oriented programming and thus introduces an object-oriented programming language, the Data Structure Manager (DSM), that complements classes with relationships. Classes in DSM can declare role names for their identification in a relationship, a concept we adopted for the specification of object collaborations. Rumbaugh also perceives relationships as sets of object tuples, however, he does not further exploit his observation. Our work, in contrast, goes further and links relationships to discrete mathematics, allowing the use of mathematical relations to define invariants, a concept more powerful than the multiplicities (cardinalities) supported in DSM. Furthermore, we support relationship members and member interposition, both concepts not present in Rumbaugh's work. The missing support for member interposition is also the reason why Rumbaugh introduces qualified relations. A qualified relation is a special instance of

a ternary relationship that allows the addition of a distinguishing attribute to one of the two participants of the relationship. With member interposition, on the other hand, the need for a ternary relationship in such cases fades as the distinguishing attribute can be interposed into the respective participant.

Albano et al. [6] develop a strongly typed object-oriented database programming language with explicit support for relationships (associations) that is specifically tailored to fit the requirements of database applications. Like our work, the language Albano et al. devised allows the declaration of relationship attributes; however, it does not support relationship methods, nor member interposition. In contrast to Rumbaugh, the authors allow programmers not only to declare multiplicities (cardinality and surjectivity) but also to indicate how these constraints must be maintained, i.e., whether to cascade an operation that endangers the constraint or whether to prevent it from being executed. Like DSM, however, the language of Albano et al. lacks support for expressing constraints other than multiplicities.

The main contribution of the work by Bierman and Wren [4] is to provide the type system and the operational semantics of a Java-like language that supports relationships. In this way, the authors describe how a strongly typed class-based object-oriented language, like Java, can be extended to support relationships. Bierman and Wren further introduce relationship inheritance, a concept not considered in this paper. Again, our work mainly differs from the work by Bierman and Wren in its support for member interposition and relationship invariants.

Pearce and Noble [7,8] show how to use aspects [26,27] to implement relationships and multiplicities in a class-based object-oriented language. In an aspect-based implementation, relationship members can then be interposed into participants through inter-type declarations.

Noble and Grundy [28] describe ways of persisting relationships from the modeling to the implementation stage in object-oriented development by transforming analysis relationships into corresponding classes. Their approach is purely class-based and does not mention language support for relationships.

Helm et al. [29] use contracts to specify the behavioral compositions in class-based object-oriented systems. Similar to relationships, these contracts allow the programmer to explicitly state which classes collaborate with each other. The focus of the work by Helm et al., however, is the specification of collaborative behavior. A contract, for example, can declare actions that need to be executed by the participants and can impose an ordering on the execution.

Aksit et al. [30] propose Abstract Communication Types (ACTs), classes describing object interactions, as a means to encapsulate these interactions at the programming language level. ACTs rely on composition filters for their integration with the remaining system and act in response to calls issued from the underlying classes that are forwarded and possibly adapted by these filters. Relationships, on the contrary, are self-contained and independent of the events happening in the participating classes.

Herrmann [31] describes a Java-like language supporting Object Teams, the modules encapsulating multi-object collaborations. The focus of Herrmann's

work is the a posteriori integration of collaborations into existing systems. The language thus allows the programmer to forward method calls from teams to base classes and offers advice-like constructs, known from aspect-oriented programming [26], to override methods of base classes.

Reenskaug [10] introduces role models to describe the structure of cooperating objects along with their static and dynamic properties. Role models are purely conceptual and focus on message-based interactions; however, they could assist in the identification of relationships during system design as relationships can be regarded as representations of particular role models.

8 Concluding Remarks

Relationships capture the collaborations between objects and provide a key to understanding large-scale object-oriented systems. This paper introduces mathematical relations as an abstraction of relationships and develops the concept of member interposition (which points out those members of classes that participate in a relationship). Once relationships are explicit in a program, it is possible to express invariants that extend beyond the inside of a class (or class instance). Invariants can be classified along two orthogonal dimensions: there are intra-relationship and inter-relationship invariants as well as value-based and structural invariants.

Understanding and reasoning about object-oriented programs remains a difficult issue, and many approaches have been suggested to help the programmer in this task. Mathematical relations as a formal model of relationships provide a solid foundation to deal with the object collaborations in such a program. Relationships that are explicit widen the view of the programmer (and ultimately the view of tools) so that it is possible to reason (and optimize) beyond class boundaries.

Acknowledgments

We thank Jean-Raymond Abrial, Peter Müller, and Laurent Voisin for their feedback.

References

1. Goldberg, A., Robson, D.: Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading (1983)
2. Jacobson, I., Booch, G., Rumbaugh, J.E.: The Unified Software Development Process. Addison-Wesley, Reading (1999)
3. Chen, P.P.S.: The entity-relationship model - toward a unified view of data. ACM Transactions on Database Systems (TODS) 1(1), 9–36 (1976)
4. Bierman, G.M., Wren, A.: First-class relationships in an object-oriented language. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 262–286. Springer, Heidelberg (2005)

5. Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In: 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87), pp. 466–481. ACM Press, New York (1987)
6. Albano, A., Ghelli, G., Orsini, R.: A relationship mechanism for a strongly typed object-oriented database programming language. In: 17th International Conference on Very Large Data Bases (VLDB'91), pp. 565–575. Morgan Kaufmann Publishers Inc. San Francisco (1991)
7. Pearce, D.J., Noble, J.: Relationship aspects. In: 5th International Conference on Aspect-Oriented Software Development (AOSD '06), pp. 75–86. ACM Press, New York (2006)
8. Pearce, D.J., Noble, J.: Relationship aspect patterns. In: 11th European Conference on Pattern Languages of Programs (EuroPLoP'06) (2006)
9. Booch, G.: The Unified Modeling Language User Guide. Addison-Wesley, Reading (1999)
10. Reenskaug, T., Wold, P., Lehne, O.A.: Working with Objects: The OOram Software Engineering Method. Manning/Prentice Hall, Englewood Cliffs (1996)
11. Bäumer, D., Riehle, D., Siberski, W., Wulf, M.: The role object pattern. In: 4th Conference on Pattern Languages of Programs (PLoP'97) (1997)
12. Gamma, E.: The extension objects pattern. In: 3rd Conference on Pattern Languages of Programs (PLoP'96) (1996)
13. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
14. Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. In: 1st ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86), vol. 21, pp. 38–45. ACM Press, New York (1986)
15. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
16. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
17. Meyer, B.: Eiffel: The Language. Prentice-Hall, Englewood Cliffs (1991)
18. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2004)
19. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for java. Technical Report 98-06-rev29, Iowa State University (2006)
20. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of jml tools and applications. International Journal on Software Tools for Technology Transfer (STTT'05) 7(3), 212–232 (2005)
21. Leino, K.R.M., Müller, P.: Modular verification of static class invariants. In: Fitzgerald, J.A., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 26–42. Springer, Heidelberg (2005)
22. Jacobs, B., Kiniry, J., Warnier, M.: Java program verification challenges. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2002. LNCS, vol. 2852, pp. 202–219. Springer, Heidelberg (2003)
23. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing (to appear, 2006)

24. Liskov, B., Zilles, S.: Programming with abstract data types. In: ACM SIGPLAN Symposium on Very High Level Languages, pp. 50–59. ACM Press, New York (1974)
25. Willis, D., Pearce, D.J., Noble, J.: Efficient object querying for java. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 28–49. Springer, Heidelberg (2006)
26. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
27. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
28. Noble, J., Grundy, J.: Explicit relationships in object-oriented development. In: Conference on the Technology of Object-Oriented Languages and Systems (TOOLS'95), pp. 211–226. Prentice-Hall, Englewood Cliffs (1995)
29. Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: specifying behavioral compositions in object-oriented systems. In: European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90), pp. 169–180. ACM Press, New York (1990)
30. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting object interactions using composition filters. In: Guerraoui, R., Riveill, M., Nierstrasz, O. (eds.) Object-Based Distributed Programming. LNCS, vol. 791, pp. 152–184. Springer, Heidelberg (1994)
31. Herrmann, S.: Object teams: Improving modularity for crosscutting collaborations. In: Aksit, M., Mezini, M., Unland, R. (eds.) NODe 2002. LNCS, vol. 2591, pp. 248–264. Springer, Heidelberg (2003)