

Relations: Abstracting Object Collaborations

Stephanie Balzer* Patrick Eugster† Thomas R. Gross‡

Technical Report 539, 2006
Department of Computer Science, ETH Zurich

Abstract

Allowing the description of a collection of objects, the concept of a “class” is central to object-oriented programming languages, yet, it is inadequate to describe the collaborations that arise from the interactions between these objects. The continued interest in concretizing object interactions — be it on the level of design through patterns, on the level of programming languages through first-class relationship support, or on the level of formal program specification through contracts — indicates that classes alone do not suffice. In this paper, we build upon first-class relationships serving as the module to encapsulate object collaborations. We introduce a novel abstraction to reason upon object collaborations: *relations*. As relationships allow the description of collections of groups of interacting objects, they can be viewed as sets of object tuples and thus as relations. The abstraction of a relation enables the specification of relationships using discrete mathematics. For example, we can express the structural characteristics of relationships by means of the mathematical properties of their defining relations.

1 Introduction

The module of a *class* is the basic unit of program decomposition in common class-based object-oriented programming languages. Classes proved suitable to model real-world entities as they allow the description of the properties of these entities by means of attributes and methods. However, various endeavors in both research and industry provide evidence that this granularity of decomposition is insufficient to model object collaborations.

Design patterns, for example, have been proposed to capture the structure of communicating classes that are customized to solve a general design problem in a particular context [10]. Design patterns have been used with great success to express design intent, they are not formal enough to allow *reasoning* about them. A number of projects therefore use *relationships* [26, 3, 5, 27, 23] to express the collaborations that exist between classes, and relationships become first-class citizens of the programming language. Relationships together with classes then constitute the primary programming language modules that

*ETH Zurich, Switzerland, stephanie.balzer@inf.ethz.ch

†Purdue University, USA, p@cs.purdue.edu

‡ETH Zurich, Switzerland, thomas.gross@inf.ethz.ch

both can declare their own members (attributes and methods). Contract-based programming and specification languages supporting invariants, such as Eiffel [19, 18], Spec# [4], and JML [15, 7] go a step further and support *contracts* to specify the mutual obligations between collaborating classes. Unfortunately, the verification of actual contract-based programs can sometimes become quite challenging [12, 16], and the difficulties point to the need for other modules to capture the collaboration(s) between objects.

In this paper, we present a mechanism for the specification of object collaborations in class-based, statically typed, object-oriented programming languages. Building upon existing efforts, we use first-class relationships as the modules encapsulating object collaborations. The main contribution of this paper is to introduce *mathematical relations* as an abstraction of relationships. Relations, being sets of tuples, lend themselves to describe sets of object tuples, with tuples representing the objects that interact with each other. The abstraction of a relation allows the specification of relationships using discrete mathematics. For example, we can express the structural characteristics of relationships by means of the mathematical properties of their defining relations. To accommodate the definition of these properties at a module level, we extend first-class relationships with *invariants*.

Similar to the class invariants known from class-based object-oriented programming and specification languages supporting contracts, relationship invariants can be viewed as the “consistency conditions” that have to hold true for all incarnations of these relationships. The artefact on which these conditions are imposed, however, is different in both approaches: whereas class invariants enable the specification of the consistency conditions holding within classes, relationship invariants target at the conditions controlling the collaboration emerging in-between classes.

The use of relations for abstracting object collaborations proves not only viable for expressing relationship invariants, but also for reasoning upon the behavior of a system composed of objects and relationship instances. Of interest, in particular, are state transitions of the system that potentially affect the validity of relationship invariants. We therefore devise an *invariant preservation model* that (i) identifies the programming language operations triggering state transitions, that (ii) declares which operations are admissible, and that (iii) defines the invariant-preserving actions for every kind of relation and operation applied.

The remainder of this paper is organized as follows: Section 2 provides a short introduction to the specification of consistency conditions of classes using class invariants and details the dependent delegate dilemma, an ongoing challenge in contract-based verification. Section 3 summarizes the main concepts of first-class relationships and defines the terminology of this paper. Section 4 discusses the applicability of relations to first-class relationships. This includes the introduction of relationship invariants and the elaboration of the invariant preservation model. Section 5 provides an overview of the related work and Section 6 concludes this paper.

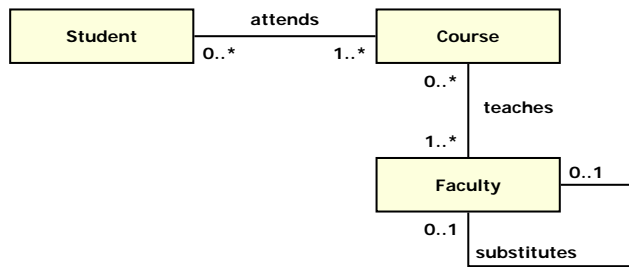


Figure 1: Simple university information system, UML diagram.

2 Challenges in contract-based verification

The verification of class invariants can sometimes become difficult. In this section, we first introduce the running example of the paper. Based on the example, we then discuss the dependent delegate dilemma, the challenging verification of class invariants in the presence of callbacks.

2.1 Simple university information system

Figure 1 depicts a Unified Modeling Language (UML) class diagram representing a greatly simplified version of the information system of a university. Variations of this example can be found in several related publications [5, 23, 6].

The UML class diagram displays the classes `Student`, `Course`, and `Faculty`. Furthermore, it models the collaborations that emerge from the interactions between these classes by means of *associations*. There are three such associations: *attends*, representing students attending courses, *teaches*, representing faculty members teaching courses, and *substitutes*, representing faculty members filling in for other faculty members. UML class diagrams further allow the specification of the *multiplicities* of associations by indicating the lower and upper bounds of the number of participating objects. For example, students are required to attend courses, whereas courses can accommodate several students, but do not have to. Similar considerations apply to the *teaches* association, with the difference that a course must have at least one faculty member assigned, whereas not all faculty members have to teach courses. The multiplicities of the *substitutes* association, finally, guarantee that every faculty member has at most one deputy.

2.2 Dependent delegate dilemma

Figure 2 shows, in Java-like pseudocode, a typical implementation of the `Faculty` class and the *substitutes* association. The implementation uses *contracts*. Contracts are supported by a number of class-based object-oriented languages, such as the Eiffel programming language [19, 18], the Spec# programming system [4], and the behavioral interface specification language for Java, JML (the Java Modeling Language) together with its supportive verification tools [15, 7]. Contracts are expressed in the form of *preconditions*, *postconditions*, and *invariants* and define correctness conditions specifying the mutual obligations of

collaborating classes [19, 9]. A violation of a precondition shows that the client of a class failed to establish the proper context, whereas a broken postcondition blames the supplier for not complying with the arrangement. Invariants, furthermore, specify consistency conditions of the program’s data that are to be maintained throughout the execution of the program.

The example uses invariants as well as preconditions. The irreflexiveness of the `substitutes` association (no faculty member can be its own deputy) is established by the invariant

```
(getDeputy() != this) &&
(getInferior() != this)
```

The asymmetry of the association (two distinct faculty members cannot substitute each other) is established by the invariant

```
(getInferior() != null && getInferior() != null
=> getDeputy() != getInferior())
```

The invariant specification

```
(getDeputy() != null
=> getDeputy().getInferior() == this) &&
(getInferior() != null
=> getInferior().getDeputy() == this)
```

may seem to be redundant at first. However, as associations are bidirectional — one object collaborates with another one and vice versa — and this bidirection is lost when representing the collaboration with unidirectional references, we need to re-introduce the bidirection by indicating both directions in the specification of an invariant.

The verification of the invariant of class `Faculty` is not straightforward. A detailed discussion of various proof techniques and proposed solutions to address this kind of problem is beyond the scope of this paper; the interested reader may refer to the related literature [12, 17, 20, 16]. Moreover, we do not attempt to suggest another proof technique; our sole interest in studying such verification challenges is to identify the particular structure of a program that makes verification troublesome.

The verification of class `Faculty` is challenging as it gives rise to the *dependent delegate dilemma* [12, 17, 20, 16]. The dilemma is also referred to as the *callback* [17, 12] or the *re-entrance* [12] problem. As the terms *callback* and *re-entrance* appear in various contexts, we use the less ambiguous term “dependent delegate dilemma” in this paper [20]. The dependent delegate dilemma potentially occurs whenever a method delegates a task to another object — the so-called *delegate* — by calling the method accomplishing the task on that delegate object. This configuration by itself is not problematic; however, the verification task becomes troublesome when in addition to the method call on the dependent delegate the invariant of the current object is temporarily violated and, furthermore, the dependent delegate later on calls back into the originating object. Temporary invariant violations during method executions are generally acceptable, provided that the invariant is re-established once the method returns. In the configuration described, though, the violation poses problems as the dependent delegate may encounter the current object in an inconsistent state.

```

class Faculty {

    private String name;
    private Location office;
    private LinkedList courses;

    // Faculty current object is deputy of
    private Faculty inferior;
    // Faculty that is the deputy of current object
    private Faculty deputy;

    invariant
    (getDeputy() != this) &&
    (getInferior() != this) &&
    (getDeputy() != null
     => getDeputy().getInferior() == this) &&
    (getInferior() != null
     => getInferior().getDeputy() == this) &&
    (getInferior() != null && getInferior() != null
     => getDeputy() != getInferior());

    void substitute(Faculty f)
    requires
    deputy == null && f.getInferior() == null &&
    f != null && f != this;
    {
        this.deputy = f;
        f.setInferior(this);
    }

    void setInferior(Faculty f)
    {
        this.inferior = f;
    }

    Faculty getInferior() {
        return inferior;
    }

    ... // Residual setter and getter methods
}

```

Figure 2: Substitutes association with contracts.

In the example (see Figure 2), the call of the dependent delegate happens in the statement

```
f.setInferior(this);
```

in method `substitute()`. In this call, a reference to the current object is passed to the callee `f`, the dependent delegate. At the moment of the call the invariant of the current object is temporarily broken as it has been destroyed in the previous statement

```
this.deputy = f;
```

This poses problems for program verification as any assumptions by the dependent delegate on the state of the originating object are compromised: the invariant might hold or be broken. It is worthwhile noting that changing the order of statements in method `substitute()` does not help either. Although an inverted order evades a violation of the invariant of the current object, it would break the invariant of the deputy object.

As mentioned, solutions to address the dependent delegate dilemma have already been proposed in the past [20, 17]. These approaches however, either result in a relaxation of the specification of the class [20], or they require the programmer to augment the specification by explicitly declaring when the invariant may be assumed and when it may not be assumed [17].

In the case of class `Faculty`, the violation of the invariant occurs because the invariant stretches across object boundaries and, as a consequence, the assignments to the attributes involved in the invariant happen in several method executions called on different objects. This clashes with the semantics of class invariants, namely that they hold both before and after the execution of a method.

Rather than adapting the semantics of class invariants, we suggest questioning the modularization of our example. In his article on modular decomposition [22], Parnas advises decomposing systems in such a way that each module hides its underlying design decisions from the others. Invariants reflect the design decisions buried in our code. Conceptually, the class `Faculty` represents two modules, with one module for every direction of the collaboration (inferior versus deputy). Both these modules share the same design decision. Following Parnas' advice, we need to fuse the two modules to encapsulate their common design decision. We can achieve this by making their collaboration explicit, using the abstraction of a relationship.

3 First-class relationships

In this section we provide a short motivation for the need of first-class relationships and introduce the terminology used in this paper. We then illustrate the concepts through an example.

3.1 Explicit representation of object collaborations

The need to explicitly represent object collaborations is increasingly agreed upon [26, 3, 21, 5, 27, 23]. Although such collaborations can be readily modeled as *associations* in UML class diagrams, for example, or *relationships* in

ER (Entity-Relationship) diagrams [8], common class-based object-oriented programming languages lack the necessary module to reflect object collaborations. Programmers consequently must use (unidirectional) references to represent collaborations (e.g. *courses* or *inferior* in Figure 2). This approximation results in the loss of the conceptual notion of a collaboration [26] and the information about a given collaboration is distributed across multiple classes [5]. Moreover, classes contain — in addition to their defining attributes and methods — the code necessary to represent and maintain collaborations; making these classes larger, less cohesive, and more complex [23].

In response, numerous authors propose to preserve relationships from the design to the implementation stage [26, 3, 21, 5, 27]. Noble and Grundy [21] show how to incorporate analysis relationships directly within a program’s code and provide evidence that programs constructed in this way tend to be smaller and easier to understand. Rumbaugh [26], Albano et al. [3], and Bierman and Wren [5] go further and require relationships to become *first-class* citizens of the programming language. Relationships together with classes then constitute the primary language modules, which can define their own members.

3.2 Terminology

In this paper we propose to use first-class relationships to capture object collaborations. We now briefly introduce our terminology.

Classes: Like in any other class-based, object-oriented approach, *classes* act as foundries for objects. They provide a description of both the data and the behavior of a collection of similar objects. Classes portray that description by means of *members*, which can be *attributes* (data) or *methods* (behavior). Each object, created by instantiating a class, exhibits specific values for its attributes, constituting the object’s state.

Relationships: *Relationships* are the modules encapsulating object collaborations. They are not meant to replace classes, but rather complement them. Like classes, relationships declare *members*, which can be *attributes* or *methods*. Relationships can also declare *invariants*, which specify the properties of the mathematical relation underlying the collaboration (see Section 4). Analogously to the differentiation between classes and instances of classes, we distinguish a *relationship*, denoting the type of a relationship, from a *relationship instance*, denoting an instance of a relationship.

Participants: Relationships describe the behavior emerging from the collaboration of objects. The *participants* of a relationship are the defining classes of the objects involved in the collaboration. The *participating objects* are the actual objects that take part in a particular relationship instance.

Roles: To indicate the direction of the relation that defines the collaboration, *roles* can be attached to the participants of relationships.

Unlike a pure object-oriented approach, the classes of a system modeled with first-class relationships do not declare collaborative behavior. The declaration of object collaborations is the domain of relationships.

Classes and relationships are orthogonal, yet the declaration of invariants might prevent classes from being instantiated outside of a relationship or limit

```

relationship Attendance {
  participants(Student attendee, Course subject);
  ...
}

relationship Instruction {
  participants(Faculty lecturer, Course lecture);
  ...
}

relationship Substitution {
  participants(Faculty inferior, Faculty deputy);
  ...
}

```

Figure 3: Relationships of the running example.

the number of objects a particular object collaborates with through a specific relationship (see Section 4.1).

For simplicity, we restrict the presentation in this paper to binary relationships (relationships with two participant classes). The concepts we present, however, are not influenced by this decision and can easily be generalized to the case of n-ary relationships.

3.3 Illustration

Modeling the university information system of Figure 1 with the modules introduced in the previous section yields the classes `Student`, `Course`, and `Faculty` and the relationships `Attendance`, `Instruction`, and `Substitution`. Figure 3 depicts the resulting relationships using a Java-like notation.

We use role names to denote the role of a participant in a relationship. For instance, in the relationship `Substitution` the first participant plays the role of the `inferior`, whereas the second plays the role of the `deputy`. Roles are particularly helpful when a relationship involves several instances of the same class. Since first-class relationships allow us to model the object collaborations explicitly in a single module, the participating classes, furthermore, become smaller, more cohesive, and less complex as they do no longer contain the code necessary to represent and maintain collaborations.

4 Relations

First-class relationships as proposed in [26, 3, 21, 5, 27, 23] do not allow a precise specification of the structure of object collaborations. The main contribution of this paper is the introduction of *mathematical relations* as an abstraction of relationships and the support of *relationship invariants* as the vehicle to specify the properties of these relations at module level. In this section, we provide a detailed discussion of how relations ease the specification and reasoning upon object collaborations.

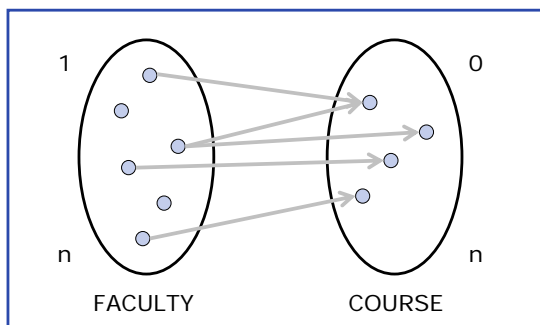


Figure 4: Graphical representation of the Instruction relationship.

4.1 Relationship invariants

Figure 4 shows a Venn diagram-like representation of the `Instruction` relationship. The small circles stand for objects and the arrows for relationship instances. The diagram nicely conveys the argument why relations serve as abstractions of relationships: class extensions can be abstracted as *sets* comprehending all the objects that potentially participate in the relationship and relationship extensions can be abstracted as *sets of object tuples*, and thus, as *relations*. The arrow heads indicate the direction of the relation.

Thanks to the abstraction of a relation, we have the full range of discrete mathematics at our disposal to specify the structural characteristics of relationships. In the case of the `Instruction` relationship, for example, we require that every course needs to have at least one lecturer assigned to it. This clearly can be expressed by a *surjective relation* (1). We thus impose this restriction as an invariant on the relationship `Instruction`. For the remaining relationships of the running example, we get a *total relation* (2) for the `Attendance` relationship and an *asymmetric, irreflexive, partial injection* (3) for the `Substitution` relationship. Note, that we use throughout the paper standard Java-like capitalization for programming language elements (e.g. classes and relationships) and all uppercase letters for their corresponding abstractions (e.g. sets and relations). Table 1 gives a short explanation on the mathematical notation used.

$$INSTRUCTION \in FACULTY \leftrightarrow COURSE \quad (1)$$

$$ATTENDANCE \in STUDENT \leftrightarrow COURSE \quad (2)$$

$$SUBSTITUTION \in FACULTY \rightsquigarrow FACULTY \quad (3)$$

$$SUBSTITUTION \cap SUBSTITUTION^{-1} = \emptyset$$

$$SUBSTITUTION \cap id(FACULTY) = \emptyset$$

Table 1: Mathematical symbols using the Event-B notation.

Symbol	Description
\mapsto	Pair constructing operator
$S \leftrightarrow T$	Set of binary relations from S to T
$S \twoheadrightarrow T$	Set of surjective relations from S to T
$S \leftrightarrow T$	Set of total relations from S to T
$S \mapsto T$	Set of partial injections from S to T
$S \mapsto T$	Set of partial functions from S to T

4.2 Invariant preservation model

In a system composed of objects and relationship instances, relationship invariants must hold true for every relationship instance and must be preserved from one state to the other along state transitions of the system. In this section, we elaborate an *invariant preservation model* that (i) identifies the programming language operations triggering state transitions, that (ii) declares which operations are admissible, and that (iii) defines the invariant-preserving actions for every kind of relation and operation applied.

The invariant preservation model we devise in the following can itself be represented and proven using *Event-B* [1], a methodology to model software systems based on discrete mathematics and refinement. We adhere to the notation introduced in [1]. Table 1 explains briefly the possibly non-standard symbols appearing in the text.

Programming language operations that cause state transitions are the creation and deletion of objects and relationship instances. Thanks to the abstraction introduced in this paper, namely that relationships can be abstracted as relations, the creation and deletion of objects and relationship instances can be modeled as simple set operations. The creation of an object a results in adding that object to the *extension* AE of its class A . Note, that we use different sets to distinguish classes from their extensions. A is the set of all *potential* objects of the class \mathbf{A} , whereas AE is the set of *existing* objects of \mathbf{A} . The two sets are related as follows: $AE \subset A, a \in A, a \notin AE \vdash AE' = AE \cup \{a\}$. The deletion of an object a , on the other hand, requires removing the object from the extension AE of its class A : $AE \subset A, a \in A, a \in AE \vdash AE' = AE \setminus \{a\}$. Similar considerations hold for relationships as well. Creating a relationship instance results in adding the corresponding object pair $a \mapsto b$ to the relation R associated with the relationship: $R \in AE \leftrightarrow BE, a \in AE, b \in BE \vdash R' = R \cup \{a \mapsto b\}$. Deleting a relationship instance requires removing the corresponding object pair $a \mapsto b$ from the relation R : $R \in AE \leftrightarrow BE, a \mapsto b \in R \vdash R' = R \setminus \{a \mapsto b\}$.

As the elements of the relations are object pairs¹, there is an interdependence between objects and relations. More precisely, we can instantiate a relationship only if both participating objects exist, and we must remove an object pair from a relation as soon as either of the participating objects becomes unreachable. Assuming the most general form of a relation, $R \in AE \leftrightarrow BE$, we can formalize this interdependence as follows:

¹Remember, that we restrict the presentation in this paper to binary relationships

$$R \in AE \leftrightarrow BE \Leftrightarrow \forall a, b. (a \mapsto b \in R \Rightarrow a \in AE \wedge b \in BE) \quad (4)$$

The kind of interdependence between objects and relations relies on the specific properties a relation exhibits. For example, in case of a partial function, it is only possible to make an object participate with another object in a relationship if the first one is not yet participating in that relationship. The interdependence is different in case a relation is total. Then, objects cannot be created without adding at least one corresponding object pair to the relation. We can formalize these interdependences as follows:

$$R \in AE \leftrightarrow BE \Leftrightarrow R \in AE \leftrightarrow BE \wedge \forall a. (a \in AE \Rightarrow \exists b. (b \in BE \wedge a \mapsto b \in R)) \quad (5)$$

$$R \in AE \leftrightarrow BE \Leftrightarrow R \in AE \leftrightarrow BE \wedge \forall a, b, c. (a \mapsto b \in R \wedge a \mapsto c \in R \Rightarrow b = c) \quad (6)$$

Tables 2 and 3 summarize the interdependences between relations and objects taking into account the range of possible properties relations may exhibit. Table 2 indicates the invariant-maintaining actions to be applied to the corresponding relations whenever objects are created or deleted and details under what circumstances the operations are admissible. Table 3 depicts how changes in participation of object pairs influence individual objects.²

A programming language supporting first-class relationships and relationship invariants must comply with the semantics defined by the invariant-preserving model. Such a language must monitor that only the invariant-preserving actions identified in Table 2 and Table 3 can be applied. This in turn guarantees the properties (4), (5), and (6).

4.3 Illustration

We illustrate the concepts introduced so far at the example of the `substitutes` association (Figure 1). The resulting `Substitution` relationship is depicted in Figure 5. As pointed out earlier, the participants of a relationship become much simpler when object collaborations are expressed explicitly using first-class relationships. In particular, all the methods for maintaining collaborations become obsolete.

In the contract-based implementation of the example (Figure 2), the call of the dependent delegate happens in method `substitute()`. Because of the use of first-class relationships, that method is no longer present in Figure 5. To be sure that the dependent delegate dilemma has definitively disappeared, we need to check whether the introduction of the invariant has no adverse effect. In the contract-based implementation, the dependent delegate dilemma occurs as the invariant stretches across object boundaries. In the current implementation the modularization is different. The new module — the relationship

²Note that the exchange of one object with another one in an object pair reduces to a disconnection followed by a connection.

Table 2: Summary of admissible operations and invariant-preserving actions in response to object creation and deletion.

Kind	Set	
	Class Extension	Relation
Relation		
Object Creation	Add object	
Object Deletion	Remove object	Remove all pairs containing object
Total Relation		
Object Creation	Add object only if pair with object at first position is added simultaneously	Add at least one pair with object at first position
Object Deletion	Remove object	Remove all pairs containing object
Partial Function		
Object Creation	Add object	Add only if at most one pair with object at first position
Object Deletion	Remove object	Remove all pairs containing object
Total Function		
Object Creation	Add object only if pair with object at first position is added simultaneously	Add only if at most one pair with object at first position
Object Deletion	Remove object	Remove all pairs containing object

```

relationship Substitution {

  participants(Faculty inferior, Faculty deputy);

  invariant
  partialInjection(inferior, deputy) &&
  asymmetric(inferior, deputy) &&
  irreflexive(inferior, deputy);

}

```

Figure 5: Substitution relationship with invariants.

Table 3: Summary of admissible operations and invariant-preserving actions in response to the manipulation of object pairs.

Kind	Set	
Operation	Relation	Class Extension
Relation		
Connect Pair	Add pair	
Disconnect Pair	Remove pair	
Total Relation		
Connect Pair	Add pair	
Disconnect Pair	Remove pair	Remove first object of pair if only pair with object at first position
Partial Function		
Connect Pair	Add only if at most one pair with object at first position	
Disconnect Pair	Remove pair	
Total Function		
Connect Pair	Add only if at most one pair with object at first position	
Disconnect Pair	Remove pair	Remove first object of pair

Substitution — completely encapsulates the bidirection of the collaboration. Since any method call on a relationship instance has as its target an object pair, assignments to attributes that are part of a relationship invariant need no longer being delegated to other objects. The dependent delegate dilemma has disappeared.

Comparing the two approaches, we can further see that first-class relationships and relationship invariants enable a more declarative style of constraint definition. They remove the need for helper methods to navigate among the participants of a collaboration, which results in less complex invariants.

5 Related work

The efforts that are closest to ours are (1) the recent work of Bierman and Wren [5] and (2) the preceding work of Rumbaugh [26]. Both the work of Bierman and Wren and of Rumbaugh are based on the idea to provide first-class support for relationships in object-oriented programming languages. Bierman and Wren’s work stems from the observation that, although natural and present in modeling languages, relationships between entities are only poorly representable in object-oriented programming languages. Starting from this, the authors devise their language, *RelJ*, which is a subset of Java extended with first-class relationships. RelJ provides means to define relationships between objects and to specify attributes and methods associated with these relationships. Similar considerations have also brought up by Rumbaugh [26]. He developed an object-oriented programming system, called *Data Structure Manager*, supporting first-class relationships. Although both the approaches of Rumbaugh, on one side, and Bierman and Wren, on the other side, are conceptually close to each other, the main contribution of Bierman and Wren’s work is a precise description of how a class-based, strongly typed, object-oriented language such as Java can be extended to support first-class relationships. They provide this description by giving both the type system and operational semantics of their language. In addition, Bierman and Wren also introduce the notion of relationship inheritance, a concept not present in Rumbaugh’s work.

The specification mechanism we propose has commonalities with the work of Bierman and Wren and the work of Rumbaugh, most apparently, the support of first-class relationships. However, our work importantly differs from their work in that we provide an appropriate abstraction — mathematical relation — to reason upon object collaborations. In addition, we introduces relationship invariants and define their semantics by means of the invariant preservation model.

Just like Bierman and Wren [5] and Rumbaugh [26], Albano et. al [3] consider first-class support of relationships (*associations*) to be vital. They propose a statically and strongly typed object-oriented database programming language supporting associations among classes. In addition, they include various constraints for controlling the participation of classes in associations, like, for example, referential constraints and surjectivity constraints. These constraints are tailored to the needs of database systems. Although similar to relationship invariants, these constraints do not have the same expressiveness as the mathematical relations underlying our invariants.

Recent work by Pearce and Noble [23] takes benefit of aspect-oriented pro-

gramming [14, 13] to provide first-class support for object collaborations. Since the code implementing object collaborations “cross-cuts” the code of the participating classes, aspects are a suitable module to capture the collaboration as a separate concern. The authors have developed a *Relationship Aspect Library (RAL)*, which assists the programmer in implementing *relationship aspects*. The Relationship Aspect Library also supports different types of relationships to accommodate multiplicity restrictions. Although similar to relationship invariants, such restrictions are less expressive than mathematical relations.

Our work also relates to programming and specification languages supporting invariants, such as Eiffel [19, 18], Spec# [4], and JML [15, 7]. The main difference is the scope of invariant definition. Whereas the invariants of current programming and specification languages are class-based, the invariants of our specification mechanism apply to relationships.

Noble and Grundy [21] also suggest making relationships explicit in object-oriented programs and describe a way of persisting relationships from the modeling to the implementation stage in object-oriented development. In contrast to our work, and also the work of Bierman and Wren [5] and Rumbaugh [26], their approach is purely class-based; they do not consider first-class support of relationships.

Riehle and Gross [25] propose a role modeling approach to design and integrate object-oriented frameworks. Their work is based on role modeling, introduced earlier by Reenskaug [24], and is mainly motivated by the observation that traditional class-based modeling fails to adequately describe collaborative behavior. The solution Riehle and Gross put forward consists in using role models to specify object collaborations. They further propose role model composition, a mechanism to describe multiple-purpose object collaborations. However, the work of Riehle and Gross remains a pure modeling approach, they abstain from providing first-class support for relationships. Moreover, they focus on framework design.

The need of encapsulating object interactions has already been pointed out by Aksit et al. [2]. The authors suggest *Abstract Communication Types (ACTs)*, classes describing object interactions. ACTs rely on *composition filters* for their integration with the remaining system. Composition filters are components that are interposed between classes, where they filter incoming and outgoing method calls. By delegating a received call to another object or by adapting the call before passing it on, composition filters interact with the surrounding components. Like Aksit et al. we consider the encapsulation of object interactions to be a legitimate goal. ACTs, however, are only reactive. They depend on calls issued from the underlying classes that are then redirected by composition filters. Relationships, on the contrary, are self-contained and independent of the events happening in the participating classes.

Herrmann [11] proposes a Java-like language supporting *Object Teams*, which represent modules encapsulating multi-object collaborations. Object Teams are first-class citizens spanning several base classes. The main focus of Herrmann’s work, however, is put on a posteriori integration of collaborations into existing systems. The language thus supports mechanisms to forward calls from teams to base classes. Teams, moreover, offer constructs akin to the *advices* known from aspect-oriented programming, [14] to override the methods of base classes.

6 Concluding remarks

We have presented in this paper a mechanism for the specification of object collaborations in class-based object-oriented programming languages. The mechanism is based on first-class relationships, the modules encapsulating object collaborations. The main contribution of this paper is to introduce mathematical relations as an abstraction of relationships. The abstraction of a relation allows the specification of the structural characteristics of object collaborations in terms of the mathematical properties of the relation defining a relationship. To accommodate the definition of these properties at module level, we extended first-class relationships with invariants. Relations prove not only viable for expressing relationship invariants, but also for devising the invariant preservation model. The model we elaborated reduces the creation and deletion of objects and relationship instances to simple set operations. By identifying the state transitions potentially putting invariants at risk and defining the invariant-preserving actions, the invariant preservation model presented in this paper prescribes the semantics of a programming language implementing the concepts introduced in this paper.

Acknowledgments

We thank Jean-Raymond Abrial for the many interesting discussions and helpful comments on this work. We also are grateful to Ling Liu, Nicholas Matsakis, Peter Müller, Joseph N. Ruskiewicz, and Laurent Voisin for their invaluable feedback.

References

- [1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [2] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *ECOOP Workshop*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer-Verlag, 1993.
- [3] A. Albano, G. Ghelli, and R. Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *VLDB*, pages 565–575, 1991.
- [4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS'04: International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.
- [5] G. M. Bierman and A. Wren. First-class relationships in an object-oriented language. In A. P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 262–286. Springer-Verlag GmbH, 2005.
- [6] G. Booch. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

- [7] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *STTT'05: International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
- [8] P. P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.
- [9] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 1–15, New York, NY, USA, 2001. ACM Press.
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [11] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In M. Aksit, M. Mezini, and R. Unland, editors, *NetObject-Days*, volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer-Verlag GmbH, Jan 2002.
- [12] B. Jacobs, J. Kiniry, and M. Warnier. Java program verification challenges. In *First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, volume 2852 of *Lecture Notes in Computer Science*, pages 202–219. Springer, 2002.
- [13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag GmbH, Jan 2001.
- [14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Longtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97 - Object-Oriented Programming: 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag GmbH, January 1997.
- [15] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for java. Technical Report 98-06-rev29, Iowa State University, 2006.
- [16] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. Technical Report 06-14, Department of Computer Science, Iowa State University, 2006.
- [17] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag GmbH, 2004.
- [18] B. Meyer. *Eiffel: The Language*. Prentice Hall Professional Technical Reference, 1991.
- [19] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall Professional Technical Reference, second edition, 1997.

- [20] B. Meyer. The dependent delegate dilemma. In M. Broy, J. Grünbauer, D. Harel, and C. Hoare, editors, *Engineering Theories of Software Intensive Systems, Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems, Marktoberdorf, Germany, from 3 to 15 August 2004*, volume 195 of *NATO Science Series II: Mathematics, Physics and Chemistry*, pages 105 – 118. Springer-Verlag GmbH, June 2005.
- [21] J. Noble and J. Grundy. Explicit relationships in object-oriented development. In B. Meyer, editor, *TOOLS'95: Conference proceedings on the Technology of Object-Oriented Languages and Systems*, pages 211–226. Prentice-Hall, 1995.
- [22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.
- [23] D. J. Pearce and J. Noble. Relationship aspects. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2006. ACM Press.
- [24] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Number ISBN 0-13-452930-8. Manning/Prentice Hall, 1996.
- [25] D. Riehle and T. R. Gross. Role model based framework design and integration. In *OOPSLA*, pages 117–133, 1998.
- [26] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 466–481, New York, NY, USA, 1987. ACM Press.
- [27] D. A. Thomas. On the next move in programming. *Journal of Object Technology*, 5(2):7–11, March-April 2006.