Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

Roles'07

Proceedings of the 2nd Workshop on

Roles and Relationships

in Object Oriented Programming,

Multiagent Systems, and Ontologies

Workshop co-located with ECOOP 2007 Berlin

July 30 and 31, 2007

Editors:
Guido Boella, Steffen Goebel, Friedrich Steimann,
Steffen Zschaler
Michael Cebulla

Bericht-Nr. 2007 – 9

ISSN 1436-9915

Roles'07

The 2^{nd} Workshop on Roles and Relationships in Object Oriented Programming, Multiagent Systems, and Ontologies

http://normas.di.unito.it/zope/roles07

Roles are a truly ubiquitous notion: like classes, objects, and relationships, they pervade the vocabulary of all disciplines that deal with the nature of things and how these things relate to each other. In fact, it seems that roles are so fundamental a notion that they must be granted the status of an ontological primitive.

The definition of roles depends on the definition of relationships. With the advent of Object Technology, however, relationships have moved out of the focus of attention, giving way to the more restricted concept of attributes or, more technically, references to other objects. A reference is tied to the object holding it and as such is asymmetric – at most the target of the reference can be associated with a role. This is counter to the intuition that every role should have at least one counter-role, namely the one it interacts with. It seems that the natural role of roles in object-oriented designs can only be restored by installing relationships (collaborations, teams, etc.) as first-class programming concepts.

By contrast, the relational nature of roles is already acknowledged in the area of Multiagent Systems, since roles are related to the interaction among agents and to communication protocols. However, in this area there is no convergence on a single definition of roles yet, and different points of view, such as agent software engineering, specification languages, agent communication, or agent programming languages, make different use of roles. Like its predecessor "Roles, an interdisciplinary perspective" (Roles'05) held at the AAAI 2005 Fall Symposium (see the website of the Symposium http://www.aaai.org/Press/Reports/Symposia/Fall/fs-05-08.php), this workshop aimed at gathering researchers from different disciplines to foster interchange of knowledge and ideas concerning roles and relationships, and in particular to converge on ontologically founded proposals which can be applied to programming and agent languages.

Program Committee

Uwe Assmann, Technische Universitaet Dresden
Colin Atkinson, Universitaet Mannheim
Matteo Baldoni, Universit di Torino
Giancarlo Guizzardi, LOA-CNR Trento
Stephan Hermann, Technische Universitaet Berlin
Pierre Kelsen, University of Luxembourg
Claudio Masolo, LOA-CNR Trento
James Odell, Intelligent Automation, inc. Rockville MD
Andrea Omicini, DEIS Universit di Bologna
Kasper Østerbye, IT University of Copenhagen
James Noble, Victoria University of Wellington
Daniel Oberle, SAP Research
Elke Pulvermueller, University of Luxembourg
Dirk Riehle, SAP Research, SAP Labs, LLC - Palo Alto, CA
Trygve Reenskaug, University of Oslo
Leendert van der Torre, University of Luxembourg
Harko Verhagen, DSV, KTH/SU

Table of contents

Relationships Define Roles, Objects Offer Them

Matteo Baldoni[1], Guido Boella[2], and Leendert van der Torre[3]

[1]Dipartimento di Informatica - Università di Torino - Italy. email: baldoni@di.unito.it
[2]Dipartimento di Informatica - Università di Torino - Italy. email: guido@di.unito.it
[3]University of Luxembourg. e-mail: leendert@vandertorre.com

Abstract. In this paper we study the interconnection between relationships and roles. We start from the patterns used to introduce relationships in object oriented languages. We show how the role model proposed in powerJava can be used to define roles in an abstract way in objects representing relationships, to specify the interconnections between the roles. Abstract roles cannot be instantiated. To participate in a relationship, objects have to extend the abstract roles of the relationship. Only when roles are implemented in the objects offering them, they can be instantiated, thus allowing another object to play those roles.

1 Introduction

The need of introducing the notion of relationship as a first class citizen in Object Oriented (OO) programming, in the same way as this notion is used in OO modelling, has been argued by several authors, at least since Rumbaugh [1]: he claims that relationships are complementary to, and as important as, objects themselves. Thus, they should not only be present in modelling languages, like ER or UML, but they also should be available in programming languages, either as primitives, or, at least, represented by means of suitable patterns.

Two main alternatives have been proposed by Noble [2] for modelling relationships by means of patterns:

- The relationship as attribute pattern: the relationship is modelled by means of an attribute of the objects which participate in the relationship. For example, the `Attend` relationship between a `Student` and a `Course` can be modelled by means an attribute `attended` of the `Student` and of an attribute `attendee` of the `Course`.
- The relationship object pattern: the relationship is modelled as a third object linked to the participants. A class `Attend` must be created and its instances related to each pair of objects in the relationship. This solution underlies programming languages introducing primitives for relationships, e.g., Bierman and Wren [3].

These two solutions have different pros and cons, as Noble [2] discusses. But they both fail to capture an important modelling and practical issue. If we consider the kind of examples used in the works about the modelling of relationships, we notice that relationships are also essentially associated with another concept: students are related to tutors or professors [3, 4], basic courses and advanced courses [4], customers buy from

sellers [5], employees are employed by employers, underwriters interact with reinsurers [2], *etc.* From the knowledge representation point of view, as noticed by ontologist like Guarino and Welty [6], these concepts are not natural kinds like person or organization. Rather, they all are *roles* involved in a relationship.

Roles have different properties than natural kinds, and, thus, are difficult to model with classes: roles can played by objects of different classes, they are dynamically acquired, they depend on other entities - the relationship they belong to and their players. Moreover, when an object of some natural type plays a certain role in a relationship, it acquires new properties and behaviors. For example, a student in a course has a tutor, he can give the exam and get a mark for the exam, another property which exists only as far as he is a student of that course.

We introduce roles in OO programming languages, in an extension of the Java programming language, called powerJava, described in [7–11]. The language powerJava introduces roles as a way to structure the interaction of an object with other objects calling their methods. Roles express the possibilities of interaction offered by the object to other ones (e.g., a `Course` offers the role `Student` to a `Person` which wants to interact with it), i.e., the methods they can call and the state of interaction. First, these possibilities change according to the class of the callers of the methods. Second, a role maintains the state of the interaction with a certain individual caller. As roles have a state and a behavior, they share some properties with classes. However, roles can be dynamically acquired and released by an object playing them. Moreover, they can be played by different types of classes. Roles in powerJava are essentially inner classes which are linked not only to an instance of the outer class, called institution, but also to an instance representing the player of the role. The player of the role, to invoke the methods of the roles it plays, it has to be casted to the role, by specifying both the role type and the institution it plays the role in (e.g., the university in which it is a student).

In [12] we add roles to the relationship as attribute pattern: the relationship is modelled as a pair of roles (e.g., attending a course is modelled by the role`Student` played by `Person` and `BasicCourse` played by `Course`) instead of a pair of links, like in the original pattern. In this way, the state of the relationships and the new behavior resulting from entering the relationship can be modelled by the fact that roles are adjunct instances with their state and behavior.

However, that solution fails to capture the coordination between the two roles, since in this pattern the roles are defined independently in each of the objects offering them (`Person` offering `BasicCourse` and `Course` offering `Student`) as we discuss in Section 3. This is essentially an encapsulation problem, raised by the presence of a relationship.

In this paper, we provide a solution to this limitation by introducing abstract roles defined by relationships and extended by roles of objects offering them. When roles are defined in the relationships, the interconnection between the roles can be specified (e.g., the methods describing the protocol the roles use to communicate). When roles are extended in the objects offering them, they can be customized to the context. Roles defined in the relationships are abstract and thus they cannot be instantiated. Roles can be instantiated only when they are extended in the objects which will participate to the relationship.

2 Roles and relationships

Relations are deeply connected with roles. This is accepted in several areas: from modelling languages like UML and ER to knowledge representation discussed in ontologies and multiagent systems.

Pearce and Noble [13] notice that relationships have similarities with roles. Objects in relationships have different properties and behavior: "behavioural aspects have not been considered. That is, the possibility that objects may behave differently when participating in a relationship from when they are not. Consider again the student-course example [...]. In practice, a course will have many more attributes, such as a curriculum, than we have shown."

The link between roles and relationships is explicit in modelling languages like UML in the context of collaborations: a classifier role is a classifier like a class or interface, but "since the only requirement on conforming instances is that they must offer operations according to the classifier role, [...] they may be instances of any classifier meeting this requirement" [14]. In other words: a classifier role allows any object to fill its place in a collaboration no matter what class it is an instance of, if only this object conforms to what is required by the role. Classification by a classifier role is multiple since it does not depend on the (static) class of the instance classified, and dynamic (or transient) in the sense above: it takes place only when an instance assumes a role in a collaboration [15].

As noticed by Steimann [16], roles in UML are quite similar to the concept of interface, so that he proposes to unify the two concepts. Instead, there is more in roles than in interfaces. Steimann himself is aware of this fact: "another problem is that defining roles as interfaces does not cover everything one might expect from the role concept. For instance, in certain situations it might be desirable that an object has a separate state for each role it plays, even for different occurrences in the same role. A person has a different salary and office phone number per job, but implementing the Employee interface only entails the existence of one state upon which behaviour depends. In these cases, modelling roles as adjunct instances would seem more appropriate."

To do this, Steimann [17] proposes to model roles as classifiers related to relationships, but such that these classifiers are not allowed to have instances. In Java terminology, roles should be modelled as abstract classes, where some behavior is specified, but not all the behavior, since some methods are left to be implemented in the class extending them. These abstract classes representing roles should be then extended by other classes in order to be instantiated. However, given that in a language like Java multiple inheritance is not allowed, this solution is not viable, and roles can be identified with interfaces only.

In this paper, we overcome the problem of the lack of multiple inheritance, by allowing objects participating to the relationship to offer roles which inherit from abstract roles related to the relationship, rather than imposing that objects extend the roles themselves. This is made possible by powerJava.

```
role Student playedby Person { int giveExam(String work); }
role BasicCourse playedby Course { void communicate(String text); }

class Person{
  String name;
  private Queue messages;
  private HashSet<BasicCourse> attended; //BasicCourses followed
  definerole BasicCourse {
    Person tutor;
    // the method access the state of the outer class
    void communicate (String text) {Person.messages.add(text);}
    BasicCourse(Person t){
      tutor=t;
      Person.attended.add(this); }//add link
    }
}
class Course {
  String code;
  String title;
  private HashSet<Student> attendees; //students of the course
  private int evaluate(String x){...}
  definerole Student {
    int number;
    int mark;
    int giveExam(String work)
      { return mark = Course.evaluate(work); }
    Student (){ Course.attendees.add(this); }}}//add link
```

Fig. 1. Relationship as attributes pattern with roles in powerJava

3 Relationship as attribute pattern with roles

We first describe how the relationship as attribute pattern can be extended with roles. Then, starting from the limitation of this new pattern, in Section 4 we define a new solution introducing abstract roles in relationships. As an example we will use the situation where a `Person` can be a `Student` and follow a `Course` as a `BasicCourse` in his curriculum. The language powerJava is described in [7–11] so we do not summarize it here again.

In [12], the relationship as attribute pattern is extended with roles by reducing the relationship not only to two symmetric attributes `attended` and `attendees` but also to a pair of roles. E.g., a `Person` plays the role of `Student` with respect to the `Course` and the `Course` plays the role of `BasicCourse` with respect to the `Person` (see Figure 1 and 2, where the UML representation is illustrated[1]).

[1] The arrow starting from a crossed circle, in UML, represents the fact that the source class can be accessed by the arrow target class.

The role `Student` is associated with players of type `Person` in the role specification (`role`), which specifies that a `Student` can give an exam (`giveExam`). Analogously, the role `BasicCourse` is associated with players of type `Course` in the role definition, which specifies that a `Course` can communicate with the attendee.

The role `Student` is implemented locally in the class `Course` and, viceversa, the role `BasicCourse` is defined locally in the class `Person`. Note that this is not contradictory, since roles describe the way an object offers interaction to another one: a `Student` represents how a `Course` allows a `Person` to interact with itself, and, thus, the role is defined inside the class `Course`. Moreover the behavior associated with the role `Student`, i.e., giving exams, modifies the state of the class including the role or calls its private methods, thus violating the standard encapsulation. Analogously, the `communicate` method of `BasicCourse`, modifies the state of the `Person` hosting the role by adding a message to the queue. These methods, in powerJava terminology, exploit the full potentiality of methods of roles, called *powers*, of violating the standard encapsulation of objects.

To associate a `Person` and a `Course` in the relationship, the role instances must be created starting from the objects offering the role, e.g. if `Course c`: `c.new Student(p)`.

When the player of a role invokes a method of a role, a power, it must be first role casted to the role. For example, to invoke the method `giveExam` of `Student`, the `Person` must first become a `Student`. To do that, however, also the object offering the role must be specified, since the `Person` can play the role `Student` in different instances of `Course`; in this case the `Course c`: `((c.Student)p).giveExam(...)`.

This pattern with roles allows to add state and behavior to a relationship between `Person` and `Course`, without adding a new class representing the relationship. The limitation of this pattern is that the two roles `Student` and `BasicCourse` are defined independently in the two classes `Person` and `Course`. Thus, there is no warranty that they are compatible with each other (e.g., they communicate using the same protocol, despite the fact that they offer the methods specified in the role specification). Moreover, we would like that all roles of a relationship can access the private state of each other (i.e., share the same namespace). However, this would be feasible only if the two roles `Student` and `BasicCourse` are defined by the same programmer in the same context. This is not possible since the two player classes `Person` and `Course` may be developed independently.

In summary, we would like:

- to define the interaction between the roles separately from the classes offering them to participate in the relationship, thus to guarantee that the interaction between the objects eventually playing the roles is performed in the desired way;
- that the roles of a relationship have access to the private state of each other to facilitate their programming;
- that the roles have also access to the private states of the objects offering them (like in powerJava) to customize them to the context.

These requirements mirror the complexities concerning encapsulation, which arise when relationships are taken seriously, as noticed by Noble and Grundy [5].

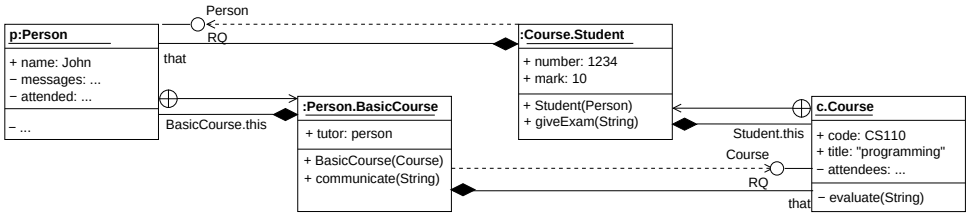


Fig. 2. The UML representation of the relationship as attributes pattern example

4 Abstract roles and relationships

A solution to the encapsulation problem is possible in powerJava by exploiting an often disregarded feature of Java. Inner classes share the namespace of the outer classes containing them. When a class extends an inner class in Java, it maintains the property that the methods defined in the inner class which it is extending continue to have access to the private state of the outer class instance containing the inner class. If the inner class is extended by another inner class, the resulting inner class belongs to the namespaces of both outer classes. Moreover, an instance of such an inner class has a reference to both outer class instances so to be able to access their states. The possible ambiguities of identifiers accessible in the two outer classes and in the superclass are resolved by using the name of the outer class as a prefix of the identifier (e.g., `Course.registry`).

This feature of Java, albeit esoteric, has a precise semantics, as discussed by [18].

The new solution we propose allows to introduce a new class representing the relationship as in the relationship object pattern, and to define the roles inside it. The idea is illustrated and in Figure 5 as an UML diagram.

First, as in the relationship object pattern, a class for creating relationship objects is created (e.g., `AttendBasicCourse`): it will contain the implementation of the roles involved in the relationship (e.g., `Student` and `BasicCourse` in `AttendBasicCourse`), see Figure 3. The interaction between the roles is defined at this level since the powers of each role can access the state of the other roles and of the relationship.

These roles must be defined as abstract and so they cannot be instantiated. Moreover, the methods containing the details about the customization of the role can be left unfinished (i.e., declared as abstract) if they need to be completed depending on the classes offering the roles which extend the abstract roles.

Second, the same roles in the relationship can be implemented in the classes offering them (and, thus, they can be extended separately), accordingly to the relationship as attribute pattern, see Figure 4 (`Person` offering `BasicCourse` and `Course` offering `Student`). However, these roles (e.g., `Student` and `BasicCourse`), rather than being implemented from scratch, extend the abstract roles of the relationship object class (e.g., `AttendBasicCourse`), filling the gaps left by abstract methods in the abstract roles (both public and protected methods). The extension is necessary to customize the roles to their new context. Methods which are declared as final in the abstract roles cannot be overwritten, since they represent the interaction among roles in

```
role Student playedby Person
  { int giveExam(String work); }
role BasicCourse playedby Course
  { void communicate(String text); }

class AttendBasicCourse {
  Student attendee;
  BasicCourse attended;
  abstract definerole Student {
    int mark;
    int number;
    //method modelling interaction
    final int giveExam(String work){
      return mark = evaluate(work);}
    //method to be implemented which is not public
    abstract protected int evaluate(String work);
  }
  abstract definerole BasicCourse {
    String program;
    Person tutor;
    //method to be implemented which is public
    abstract void communicate(String text);
  }
  AttendBasicCourse(String pr, Person t){
    attendee = c.new Student(p,this);
    attended = p.new BasicCourse(c,this,t);
  }
}
```

Fig. 3. Abstract roles

the scope of the relationship. Further methods can be declared, but they are not visible from outside since both the abstract role and the concrete one have the signature of the role declaration.

Note that the abstract roles are not extended by the classes participating in the relationship (e.g., `Course` and `Person`), but by roles offered by (i.e., implemented into) these classes (e.g., `Student` and `BasicCourse`). Otherwise, the classes participating in the relationship could not extend further classes, since Java does not allow multiple inheritance, thus limiting the code reuse possibilities.

The advantage of these solution is that roles can share both the namespace of the relationship object class and the one of the class offering the roles, as we required above. This is possible since extending a role implementation is the same as extending an inner class in Java: roles are compiled into inner classes by the powerJava precompiler.

Basing on this idea we propose here a limited extension of powerJava, which allows to define abstract roles inside relationship object classes, and to let standard roles extend them. The resulting roles will belong both to the namespace of the class offering

```
class Course {
  String code;
  String title;
  private HashSet<Student> attendees;
  class Student extends AttendBasicCourse.Student {
    Student() {
      Course.this.attendee = this;
    }
    //abstract method implementation
    protected int evaluate(String work)
      { /*Course specific implementation of the method */  }  } }

class Person {
  String name;
  private Queue messages;
  private HashSet<BasicCourse> attended;
  //courses followed as BasicCourse
  class BasicCourse extends AttendBasicCourse.BasicCourse {
    BasicCourse(Person t) {
      tutor=t;
      Person.this.attended=this; }
      //abstract method implementation
      void communicate (String text)
        {Person.this.messages.add(text);} } }
```

Fig. 4. Abstract roles extended

them and to the relationship object class. Moreover, the resulting roles will inherit the methods of the abstract roles.

Note that the abstract roles cannot be instantiated, so that the are used only to implement both the methods which define the interaction among the roles, and the methods which are requested to be contextualized. The former will be final methods which are inherited, but which cannot be overwritten in the eventual extending role: they will access the state and methods of the outer class and of the sibling roles. The latter will be abstract protected methods, which are used in the final ones, and which must be implemented in the extending class to tailor the interaction between the abstract role and the class offering the role. If these methods are declared as protected they are not visible outside the package. These methods have access to the class offering the extending roles.

Besides adding the property `abstract` to roles, three other additions are necessary in powerJava.

First, we add an additional constraint to powerJava: if a role implementation extends an abstract role, it must have the same name. Thus, the abstract and concrete role have the same requirements. Moreover, it is possible to extend only abstract roles, while general inheritance among roles is not discussed here.

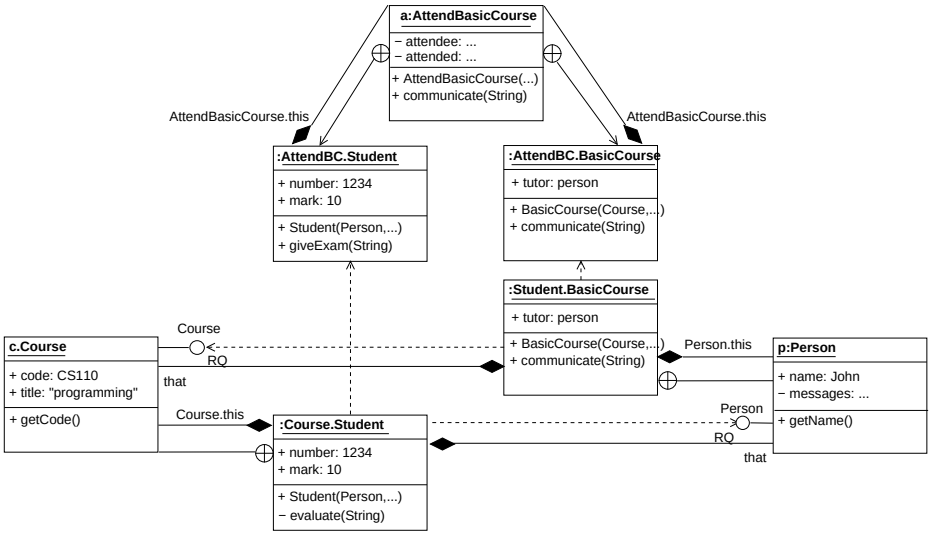


Fig. 5. The UML representation of the new relationship pattern

Second, the methods of the abstract role can make reference to the outer class of the extending role. This is realized by means of a reserved variable `outer`, which is of type `Object` since it is not possible to know in advance which classes will offer the extended role. This variable is visible only inside abstract roles.

Third, to create a role instance it is necessary to have at disposal also the relationship object offering the abstract roles, and the two roles must be created at the same time.

For example, the constructor of a relationship:

```
AttendBasicCourse(Person p, Course c){
  ...
  c.new Student(p,this);
  p.new BasicCourse(c,this);
}
```

Where `Student` and `BasicCourse` are the class names of the concrete roles implemented in `p` and `c` and they are the same as the abstract roles defined in the relation.

The types of the arguments `Person` and `Course` are the requirements of the roles `Student` and `BasicCourse` which will be used to type the `that` parameter referring to the player of the role.

Moreover, the first and the second argument of the constructor are added by default: the first one represents the player of the role, while the second one, present only in roles extending abstract roles, is the reference to the relationship object. This is necessary since the inner class instance represented by the role has two links to the two outer class instances it belongs to. This reference is used to invoke the constructor of the

abstract role, as required by Java inner classes. For example, the constructor of the role `Course.Student` is the following one.

```
Student(Person p, AttendBasicCourse a){
  a.super();
  ... }
```

However, these complexities are hidden by powerJava which adds the necessary parameters and code during precompilation.

The entities related by the relationship must preexist to it:

```
Person p = new Person();
Course c = new Course();
AttendBasicCource r = new AttendBasicCourse(p,c);
((c.Student)p).giveExam(w);
((p.BasicCourse)c).communicate(text);
```

Note that the role cast `((r.Student)p)` is equivalent to `((c.Student)p)`.

5 Conclusion

In this paper we discuss how abstract roles can be introduced when relationships are modelled in OO programs: first abstract roles are defined in the relationship object class, which specify the interaction, and then the abstract roles are extended in the classes offering them. This pattern solves the encapsulation problems raised when relationships are introduced in OO.

We introduce abstract roles using the language powerJava, a role endowed version of Java (`http://www.powerjava.org`) [7–12].

References

1. Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In: Procs. of OOPSLA. (1987) 466–481
2. Noble, J.: Basic relationship patterns. In: Pattern Languages of Program Design 4. Addison-Wesley (2000)
3. Bierman, G., Wren, A.: First-class relationships in an object-oriented language. In: Procs. of ECOOP. (2005) 262–286
4. Albano, A., Bergamini, R., Ghelli, G., Orsini, R.: An object data model with roles. In: Procs. of Very Large DataBases (VLDB'93). (1993) 39–51
5. Noble, J., Grundy, J.: Explicit relationships in object-oriented development. In: Procs. of TOOLS 18. (1995)
6. Guarino, N., Welty, C.: Evaluating ontological decisions with ontoclean. Communications of ACM **45(2)** (2002) 61–65
7. Baldoni, M., Boella, G., van der Torre, L.: Roles as a coordination construct: Introducing powerJava. Electronic Notes in Theoretical Computer Science **150** (2006) 9–29
8. Baldoni, M., Boella, G., van der Torre, L.: powerJava: ontologically founded roles in object oriented programming language. In: Procs. of OOPS Track of ACM SAC'06, ACM (2006) 1414–1418

9. Baldoni, M., Boella, G., van der Torre, L.W.N.: Modelling the interaction between objects: Roles as affordances. In: Procs. of Knowledge Science, Engineering and Management, KSEM'06. Volume 4092 of LNCS., Springer (2006) 42–54
10. Baldoni, M., Boella, G., van der Torre, L.: Interaction among objects via roles: sessions and affordances in powerjava. In: Procs. of PPPJ '06, New York (NY), ACM (2006) 188–193
11. Baldoni, M., Boella, G., van der Torre, L.: Interaction between Objects in powerJava. Journal of Object Technology **6** (2007) 7–12
12. Baldoni, M., Boella, G., van der Torre, L.: Relationships meet their roles in object oriented programming. In: Procs. of the 2nd International Symposium on Fundamentals of Software Engineering 2007 Theory and Practice (FSEN '07). (2007)
13. Pearce, D., Noble, J.: Relationship aspects. In: Procs. of AOSD. (2006) 75–86
14. OMG: OMG Unified Modeling Language Specification, Version 1.3. (1999)
15. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley (1999)
16. Steimann, F.: A radical revision of UML's role concept. In: Procs. of UML2000. (2000) 194–209
17. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data and Knowledge Engineering **35** (2000) 83–848
18. Smith, M., Drossopoulou, S.: Inner classes visit aliasing. In: ECOOP 2003 Workshop on Formal Techniques for Java-like Programming. (2003)

Member Interposition: How Roles Can Define Class Members

Stephanie Balzer and Thomas R. Gross

Department of Computer Science, ETH Zurich

Abstract. *Explicit relationships* are a means to make the collaborations that arise between objects explicit[1]. In [1], we introduce a mathematical model that fosters the specification of such relationships. The model relies in particular on *member interposition*, a mechanism that facilitates the specification of relationship-dependent members of classes. In this position paper we highlight the interdependence between relationships and *role models* and introduce *generic relationships*, an extension to support *explicit roles*.

1 Relationships and Member Interposition

A *relationship* is a programming language abstraction that encapsulates the collaborative behavior between classes. The specification of a relationship traditionally involves the indication of the participating classes and the declaration of further attributes and methods defining the collaboration. In a university information system, e.g., we have the classes `Student`, `Course`, and `Faculty`, and the relationships `Attends` (between `Student` and `Course` for students taking courses as part of their education), `Assists` (between `Student` and `Course` for students assisting courses as teaching assistants), and `Teaches` (between `Faculty` and `Courses` for faculty members teaching courses).

Member interposition is a mechanism accommodating relationship-dependent properties of classes. Such properties, e.g., the attribute `instructionLanguage`, arise only if the object participates in a specific collaboration, e.g., if a student assists a course, and therefore should not be attributed to the participating class in general. Declaring such properties as *interposed* members, on the other hand, makes the dependence explicit. Technically, the member (e.g., `instructionLanguage`) is declared as part of and considered to be part of the relationship (e.g., `Assists`) into which the member is interposed.

2 Generic Relationships and Explicit Roles

Describing the collaborations between objects, relationships can serve as the programming language counterpart of the *role models* [2] identified during system design. Likewise, a class that participates in a relationship (a *participant* of the

[1] See [1] for an introduction to the field and for a comprehensive list of related work.

relationship) can be perceived as playing the corresponding *role* defined in the model.

Unfortunately, current languages supporting relationships do not distinguish between classes and roles: a role is tied to a particular participant of a relationship. This "direct wiring" makes it difficult to accommodate variations of relationships. Even if a relationship remains identical with regard to its behavior, as soon as different participating classes are involved, a separate relationship must be declared redundantly for each participating class.

We introduce the notion of a *role* as a language construct that allows the explicit representation of the role classes can play in a relationship. The notion of a role can then be used also for the declaration of a relationship: relationships can not only list classes but also roles as their participants. Explicit roles thus enable *generic* relationship declarations, with the role serving as a generic parameter.

3 Concluding Remarks

Member interposition allows us to separate the general properties of a class from the properties that depend on the collaborations an instance of the class may take part in. Together with the support for generic relationships and the support for roles, respectively, member interposition allows the explicit representation of the roles a class may assume.

Member interposition further promotes a kind of "laziness" and "locality": the declaration of relationship-dependent members is deferred until the moment the role of the class becomes apparent, and the declaration is local to the role declaration. The benefits of laziness and locality can be exploited also for the specification and verification of software systems. In our work [1], we strictly distinguish between the definition of object collaborations and their application. Whereas class and relationship declarations allow the definition of object collaborations, a module called *application* provides the place to combine these two and to express properties that the system is supposed to maintain. The module application is a pure configurational unit that is used to define how a specific software system is composed of classes, roles, and relationships, as well as constraints that define the configuration.

References

1. Stephanie Balzer, Thomas R. Gross, and Patrick Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *21st European Conference on Object-Oriented Programming (ECOOP'07)*, Lecture Notes in Computer Science. Springer, 2007. To appear.
2. Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working with Objects: The OOram Software Engineering Method*. Number ISBN 0-13-452930-8. Manning/Prentice Hall, 1996.

Roles and Self-Reconfigurable Robots

Nicolai Dvinge, Ulrik P. Schultz, and David Christensen

Maersk Institute
University of Southern Denmark

Abstract. A self-reconfigurable robot is a robotic device that can change its own shape. Self-reconfigurable robots are commonly built from multiple identical modules that can manipulate each other to change the shape of the robot. The robot can also perform tasks such as locomotion without changing shape. Programming a modular, self-reconfigurable robot is however a complicated task: the robot is essentially a real-time, distributed embedded system, where control and communication paths often are tightly coupled to the current physical configuration of the robot. To facilitate the task of programming modular, self-reconfigurable robots, we have developed a declarative, role-based language that allows the programmer to associate roles and behavior to structural elements in a modular robot. Based on the role declarations, a dedicated middleware for high-level distributed communication is generated, significantly simplifying the task of programming self-reconfigurable robots. Our language fully supports programming the ATRON self-reconfigurable robot, and has been used to implement several controllers running both on the physical modules and in simulation.

1 Introduction

A self-reconfigurable robot is a robot that can change its own shape. Self-reconfigurable robots are built from multiple identical modules that can manipulate each other to change the shape of the robot [9, 11, 13, 4, 18, 12]. The robot can also perform tasks such as locomotion without changing shape. Changing the physical shape of a robot allows it to adapt to its environment, for example by changing from a car configuration (best suited for flat terrain) to a snake configuration suitable for other kinds of terrain. Programming self-reconfigurable robots is however complicated by the need to (at least partially) distribute control across the modules that constitute the robot and furthermore to coordinate the actions of these modules. Algorithms for controlling the overall shape and locomotion of the robot have been investigated (e.g. [5, 16]), but the issue of providing a high-level programming platform for developing controllers remains largely unexplored. Moreover, constraints on the physical size and power consumption of each module limits the available processing power of each module.

In this paper, we present a role-based approach to programming a controller for a distributed robot system. We have implemented a prototype role-based programming language, named "RAPL", for the ATRON modular, self-reconfigurable robot [9, 10]. Our implementation is based on roles as the main

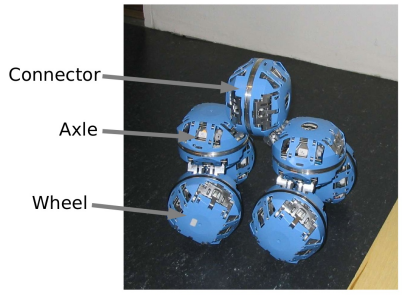


Fig. 1. The ATRON self-reconfigurable robot. Seven modules are connected in a car-like structure.

abstraction of behavior and implements a remote method invocation framework based on the roles and their structural interconnections. RAPL allows the programmer to construct a controller for a structure of ATRON modules in less time and with less knowledge of hardware than was the case before. Moreover, we argue that the controller constructed is less error-prone and is more intuitive to comprehend.

The contributions of our work are as follows: We use the concept of role-based programming to identify the central entities in a distributed robot controller. We increase the level of abstraction in the process of programming robot controllers by means of a domain specific language (DSL), which is built upon these concepts. The structural information from the DSL is used to provide a lightweight remote method invocation framework appropriate for the physical constraints of the ATRON modules. Moreover, our language allows the programmer to specify the behavior of the robot as a whole from the constituent parts. Our compiler generates an application framework either in embedded C code for execution on the physical modules or in well-structured Java code for execution in a virtual simulation environment.

2 The ATRON Self-Reconfigurable Robot

The ATRON self-reconfigurable robot is a 3D lattice-type robot [9, 10]. Figure 1 shows an example ATRON car robot built from 7 modules. Two sets of wheels (ATRON modules with rubber rings providing traction) are mounted on ATRON modules playing the role of an axle; the two axles are joined by a single module playing the role of "connector." As a concrete example of self-reconfiguration, this car robot can change its shape to become a snake (a long string of modules); such a reconfiguration can for example allow the robot to traverse obstacles such as crevices that cannot be traversed using a car shape.

An ATRON module has one degree of freedom, is spherical, is composed of two hemispheres, and can actively rotate the two hemispheres relative to each other. A module may connect to neighbor modules using its four actuated male

and four passive female connectors. The connectors are positioned at 90 degree intervals on each hemisphere. Eight infrared ports, one below each connector, are used by the modules to communicate with neighboring modules and sense distance to nearby obstacles or modules. A module weighs 0.850kg and has a diameter of 110mm. Currently 100 hardware prototypes of the ATRON modules exist. The single rotational degree of freedom of a module makes its ability to move very limited: in fact a module is unable to move by itself. The help of another module is always needed to achieve movement. All modules must also always stay connected to prevent modules from being disconnected from the robot. They must avoid collisions and respect their limited actuator strength: one module can lift two others against gravity.

Programming the ATRON robot is complicated by the distributed, real-time nature of the system coupled with limited computational resources and the difficulty of abstracting over the concrete physical configuration when writing controller programs. General approaches to programming the self-reconfigurable ATRON robot include metamodules [5], motion planning and rule-based programming. In the context of this article, we are however interested in role-based control. Role-based control is an approach to behavior-based control for modular robots where the behavior of a module is derived from its context [17]. The behavior of the robot at any given time is driven by a combination of sensor inputs and internally generated events. Roles allow modules to interpret sensors and events in a specific way, thus differentiating the behavior of the module according to the concrete needs of the robot.

3 A Role-based Conceptual Model

The level of abstraction offered by focusing on the behavior of a specific module in a given context is somewhat similar to that of role-based programming [15]. We use this approach as a basis for our language by making roles the fundamental concept for expressing the desired behavior. A fundamental difference between previous work and our approach is however that previous role-based experiments have focused on performing cyclic behavior, e.g., locomotion, and not event routing and reactive behavior. Moreover, all implementations presented in earlier work have been constructed in an ad-hoc manner with little or no language support.

Our conceptual view of a role is that it defines the module structure and the active and reactive behavior of each module in a robot. In other words, it defines what the module *can* do and what it *will* do. There is a one-to-one mapping between a role and a module, but modules can change their roles (and thus their behavior) as a reaction to messages from other modules or internal events. The behavior of a role is thus determined by the physical state of the module (as reported by sensors), program state stored in the memory of the module, and messages received from other modules. The behavior is encapsulated in methods that are activated either through external messages or internal events.

An ATRON robot as a whole is implicitly assigned a role using the object-oriented concept of a whole-part structure (as known from the whole-part design pattern [3]). Behavior for the robot is declared for each individual role. For example, all modules in a car may be able to play the role of a "car" by receiving messages for the "car" role. A module may either process such a message or forward it to another module, as designated by the programmer. The functionality of the whole and the role that it can play is thus created in coordination between the individual modules, corresponding to how the control of a modular robot necessarily must be implemented in practice.

4 The RAPL Compiler

We have implemented a role specification language for the ATRON modules, named RAPL (Role-based ATRON Programming Language). RAPL can be compiled either to Java, for use with the ATRON simulator, or to C, for execution directly on the modules. The Java backend simply generates an implementation of the proxy and state design patterns [7]. The C backend compiles a role to a skeleton that invokes C functions written by the programmer and to a proxy represented as a collection of C functions that can be used to send messages to other modules implementing this role.

4.1 RMI based communication

RAPL implements a remote procedure calling functionality to facilitate distributed communication. A RAPL program declares a list of functions each belonging to a specific role. Functions can be tied to an event or provide a default behavior for a role; an event can be a message from a neighbor-module or an internal event signal (e.g. timer, tilt sensor, etc.). Having a list of functions for each role is sufficient to generate a stub/skeleton proxy framework that provides abstraction over communication which is critical to our approach. The issue of addressing the correct modules is resolved by exploiting the structural information from the roles. Messages always identify both the receiver role and the name of the message (at runtime each identifier is represented by a single byte).

4.2 Whole-part design architecture

The whole-part design pattern is integrated in the implementation of the RMI framework, providing a whole-part functionality for the entire structure of ATRON modules. Functions declared on a controller level expose the controllers main functionality to other controllers or external clients, which would not normally know, e.g., how to make the car drive or turn.

4.3 A domain specific language with simple logic

RAPL is a domain-specific language used to express roles and functions; allowing control structures and other imperative language constructs would hamper the intention of defining behavior at a higher abstraction level. Moreover, we believe that general-purpose languages such as Java and C are better suited for specifying more complex, internal behavior in controllers. To facilitate practical experiments, we do however enable RAPL to directly express primitive actuation operations and message forwarding with simple arithmetic processing of arguments. More advanced functionality will have to be included from externally linked code supplied directly by the programmer. To this end, we have defined a simple programmatic interface between RAPL and each of the platforms that it supports: RAPL methods can be implemented in the target language and code written in the target language can call RAPL methods. We currently use XML as the concrete syntax for writing RAPL declarations; in the future we envision providing one or more high-level syntaxes, as exemplified in Figure 2.

5 Example

We now outline a few simple examples of using RAPL to program an ATRON car. We refer to the first author's MS for more detailed examples [6].

5.1 A simple car program

As a concrete example, consider the ATRON car shown earlier in Figure 1. Simple reactive control of this robot can be implemented using the role declarations shown in Figure 2, left. A role has a name and declares its structural dependencies on other roles, and can moreover extend another role creating a hierarchy of roles (not shown). Structure is specified in terms of the roles of the neighboring modules and the physical communication port used to contact the module (auto-detection of neighboring modules, although possible, is currently slightly problematic on the physical modules due to hardware difficulties). Behaviors are simply declared as functions that are attached to roles.

In the program of Figure 2, when a `move` event is delivered to the "connector" it forwards it to the two axles, which again forwards it to the wheels. The action performed by the wheel is a primitive actuation of the main joint, implemented by all modules (similarly to the methods provided by `Object` in Java). Similarly for the `turn` event. To increase readability, we are currently investigating a more human-friendly syntax resembling Java declarations, as shown in Figure 2, right.

The roles thus provide a means of denoting the behavior of each module as it is used for a specific purpose in the robot. Moreover, the roles provide a simple and very light-weight way of (albeit manually) routing events through the topology of the robot. This is particularly interesting given the resource constraints of the ATRON module, which only has 4K of RAM available for communication buffers, operating system functionality, and program state. (Program size on the other hand is not so much of a problem since there is 128K of flash memory available for storing programs.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Controller>
<Controller Name="Car" xsi="Xatron.xsd">
 <Role RoleName="Connector" ...>
  <Structure RoleToUse="Axle"
         StructureName="axleFront"
               Channel="2"/>
  <Structure RoleToUse="Axle"
         StructureName="axleRear"
               Channel="6"/>
 </Role>
 <Role RoleName="Axle" ...> ...</Role>
 <Role RoleName="Wheel" ...> ...</Role>
 <Function FunctionName="move" Target="Connector">
  <Action ActionName="move"
              Target="axleFront"
               Value="value"/>
  ...
 </Function>
 <Function FunctionName="turn" Target="Connector">
  <Action ActionName="rotate"
              Target="axleFront"
               Value="value/2"/>
  <Action ActionName="rotate"
              Target="axleRear"
               Value="value/2"/>
 </Function>
 ...
</Controller>
```

```
role Connector implements Car {
  Axle frontAxle = Axle(channel#2);
  Axle rearAxle = Axle(channel#6);

  move(int value) {
        frontAxle.move(value);
        rearAxle.move(value);
  }
  turn(int value) {
        frontAxle.rotate(value/2);
        rearAxle.rotate(-value/2);
  }
}

role Axle implements Car {
  Wheel leftWheel = Wheel(channel#0);
  Wheel rightWheel = Wheel(channel#2);

  move(int value) {
        leftWheel.move(value);
        rightWheel.move(-1*value);
  }
}

role Wheel implements Car {
  Axle axle = Axle(channel#5);
}
```

Fig. 2. A simple car controller implemented in RAPL (left) and in our proposed Java-like syntax (right)

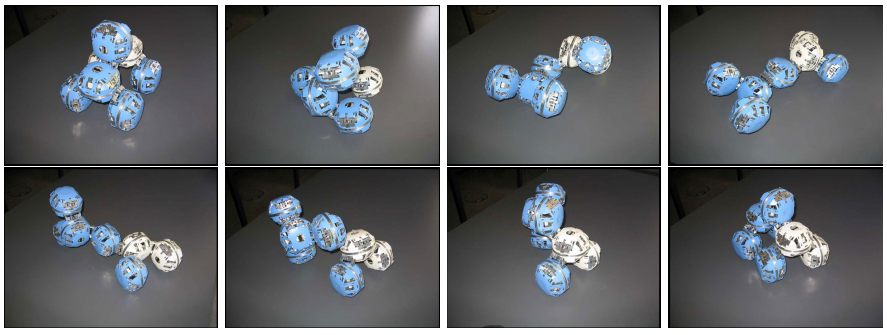


Fig. 3. The ATRON car rebuilding itself after a tilt

```
<Function FunctionName="Run" Target="Connector">
 <Action ActionName="extTiltTurnLogic" Target="EXT" Value="0"/>
 <Action ActionName="extTiltGetupLogic" Target="EXT" Value="0"/>
</Function>
<Function FunctionName="drive" Target="Car">
 <Action ActionName="move" Target="connector" Value="value"/>
</Function>
<Function FunctionName="getup" Target="Car">
 <Action ActionName="getupimpl" Target="connector" Value="value"/>
</Function>
```

Fig. 4. A tilt-aware Car controller in RAPL

```
#include "rapl.h" /* header file generated by RAPL for the car */
int last = 0;
void extTiltLogic() {
 signed char x = getTiltY();
 if(abs(x-last)>10) {
  if ((x - last) < 0) Connector_turn_impl(-20);
  else Connector_turn_impl(20);
  last = x;
 }
}
```

Fig. 5. Custom code for the tilt-logic

5.2 Adding proactive behavior

Our working example of a car is shown in action in Figure 3. Here, we have added custom functionality which fires events based on the internal tilt-sensor. Tilting makes the car turn towards higher grounds (the car will move towards the top of a hill) whereas an extreme tilt-level (indicating that the car has fallen over) triggers a series of reconfigurations which rises the car. The RAPL declarations for this more advanced controller extend those of Figure 2 and are shown in Figure 4. In this example we specify several functions and some custom code functionality to exemplify the diversity of the RAPL language.

The behavior of the controller for the tilting car is defined in the `Run` method on the connector. This tells the connector module to call the two external tilt-functions, one of which is shown in Figure 5. These two functions monitor the y-axis tilt sensor for a minor or severe change in tilt level. The first function `extTiltLogic` reacts upon tilt changes in steps of 10, and for each step it calls the `Connector` method `turn`, thus turning the axles. A complete tilt of the car (e.g., falling over) is detected by another function which triggers a self-reconfiguration sequence of several rotations. The sequence of rotations is programmed in RAPL in the function `getupimpl` (not shown). The function in the custom code of Figure 5 displays an obvious example of functionality that would be laborious to provide in our RAPL compiler and thus should be supplied in native code.

5.3 Additional examples

In addition to the car example presented thus far, we have also implemented a controller for *metamodules:* metamodules are a control approach where three modules are combined into a logical module which has the freedom to move on its own. A central module plays the role of a head whereas the two others are attached as legs, which maps perfectly to a set of RAPL declarations. Our experiments show that using RAPL the size of the controller is reduced from approximately 300 LOC in C to 32 LOC in RAPL.

Locomotion is a classical application for modular self-reconfigurable robots, and we believe RAPL will prove highly useful in this area. For example a snake structure which is simply a line of modules is easily constructed by an end-role and a body-role. In the concrete case of the ATRONs, a snake of 7 modules can transform into a car. Transformation is the main feature of a self-reconfigurable robot, but it is also the hardest part to program since modules change their physical position and most likely their role during the transformation. We envision implementing transformations using intermediate roles that are responsible for rearranging the modules before going back to a reactive state again.

We believe aggregation of roles to be essential for the scalability of our approach to more complex structures. In the work of Stoy et al, a lizard-like structure with four legs is programmed using roles [17, 15]. We observe that this scenario is close to our car-example, the main difference being legs instead of wheels. This brings to mind reuse of existing code (we can keep the central part of our structure) and aggregation of roles. If the legs of the lizard were made of metamodules implementing a whole-behaviour, the top of the car would not need to deal with how the structure actually performed its locomotion: it would simply send the same message `move` to the `Leg` or the `Wheel`.

6 Related Work

Autonomous robots are commonly controlled using behavior-based control [2]; behaviors are typically sensor-driven, reactive, and goal-oriented controllers. Certain behaviors may inhibit other behaviors, allowing the set of active behaviors to vary. Modular robots often use the concept of a role albeit in an ad-hoc fashion: complex overall behaviors can be derived from a robot where different modules react differently to the same stimuli, in effect allowing each module to play a different role (e.g., [1, 5, 14]). Recently, Stoy et al have explicitly used the concept of a role to obtain a very robust and composable behavior [15, 17]. Compared to RAPL, the implementation of roles is ad-hoc and the only control examples investigated are cyclic, signal-driven behaviors for locomotion.

Apart from RAPL, the only high-level programming language for modular robots that the authors are aware of is the Phase Automata Robot Scripting Language (PARSL) [8, 19]. Here, XML-based declarations are used to describe the behavior of each module in the PolyBot self-reconfigurable robot [18]. Compared to RAPL, the tool support is much more complete and the language has

many advanced features for controlling locomotion using behavior-based control. Nevertheless, PARSL completely lacks the concept of a role for structuring the code: each behavior is assigned to a specific module as an atomic unit.

7 Discussion and Future Work

In this paper, we have presented the RAPL system, a role-based approach to abstraction of hardware and communication in the ATRON system. Based on our experiments, we conclude that supporting role-based programming at the language level makes programming distributed robots less tedious and thereby more accessible and less error prone. We believe that providing a high-level programming interface is critical for improving the maturity of the ATRON robotic system and is an important first step in moving towards concrete applications of self-reconfigurable robots. In terms of language design, we are interested in creating a correspondence between the physical modules and the concepts used to program the controller. Here roles are a perfect fit since the behavior of a module changes over time in response to programmatic decisions or sensory input. Nevertheless, we believe that the role-based approach can be strengthened using object-oriented design principles such as whole-part.

As mentioned earlier in the context of reuse between car and lizard robots, aggregation could be used to construct "wholes of objects" which again could be parts of a bigger whole. This scenario however currently has to be programmed manually by routing messages for the whole to the appropriate module. Moreover, we would like to explore inheritance between roles, which can be realized using the standard object-oriented principles, e.g., a combination of adding and overriding functions. Currently we only support a single level of inheritance from a primitive module where basic behaviour is defined for all modules. Specialization does however raise the issue of role selection. The process of role-selection is currently done programmatically or even manually using a PDA with an infrared port. We envision automating this process to a certain degree by implementing simple rules that fire a role-change when some local state is met. This can be done right now in the custom-code section, but a future enhancement to our contribution could incorporate this logic in RAPL.

References

1. H. Bojinov, A. Casal, and T. Hogg. Multiagent control of self-reconfigurable robots. In *Proceedings of Fourth International Conference on MultiAgent Systems*, pages 143–150, 2000.
2. R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23, March 1986.
3. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.
4. A. Castano and P. Will. Autonomous and self-sufficient conro modules for reconfigurable robots. In *Proceedings of the 5th International Symposium on Distributed Autonomous Robotic Systems (DARS)*, pages 155–164, Knoxville, Texas, USA, 2000.

5. D.J. Christensen and K. Støy. Selecting a meta-module to shape-change the ATRON self-reconfigurable robot. In *Proceedings of IEEE International Conference on Robotics and Automations (ICRA)*, pages 2532–2538, Orlando, USA, May 2006.
6. Nicolai Dvinge. A programming language for ATRON modules. Master's thesis, University of Southern Denmark, 2007.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
8. Alex Golovinsky, Mark Yim, Ying Zhang, Craig Eldershaw, and Dave Duff. Polybot and PolyKinetic system: A modular robotic platform for education. In *IEEE International Conference on Robots and Automation (ICRA)*, 2004.
9. M. W. Jorgensen, E. H. Ostergaard, and H. H. Lund. Modular ATRON: Modules for a self-reconfigurable robot. In *Proceedings of IEEE/RSJ International Conference on Robots and Systems (IROS)*, pages 2068–2073, Sendai, Japan, September 2004.
10. H.H. Lund, R. Beck, and L. Dalgaard. Self-reconfigurable robots with ATRON modules. In *Proceedings of 3rd International Symposium on Autonomous Minirobots for Research and Edutainment (AMiRE 2005)*, Fukui, 2005. Springer-Verlag.
11. S. Murata, E. Yoshida, K. Tomita, H. Kurokawa, A. Kamimura, and S. Kokaji. Hardware design of modular robotic system. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2210–2217, Takamatsu, Japan, 2000.
12. D. Rus and M. Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *Journal of Autonomous Robots*, 10(1):107–124, 2001.
13. W.-M. Shen, M. Krivokon, H. Chiu, J. Everist, M. Rubenstein, and J. Venkatesh. Multimode locomotion via superbot robots. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, pages 2552–2557, Orlando, FL, 2006.
14. Wei-Min Shen, Yimin Lu, and Peter Will. Hormone-based control for self-reconfigurable robots. In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*, pages 1–8, New York, NY, USA, 2000. ACM Press.
15. Kasper Stoy, Wei-Min Shen, and Peter Will. Using role based control to produce locomotion in chain-type self-reconfigurable robots. *IEEE Transactions on Robotics and Automation, special issue on self-reconfigurable robots*, 2002.
16. K. Støy. How to construct dense objects with self-reconfigurable robots. In *Proceedings of European Robotics Symposium (EUROS)*, pages 27–37, Palermo, Italy, May 2006.
17. K. Støy, W.-M. Shen, and P. Will. Implementing configuration dependent gaits in a self-reconfigurable robot. In *Proceedings of the 2003 IEEE international conference on robotics and automation (ICRA'03)*, pages 3828–3833, Tai-Pei, Taiwan, September 2003.
18. M. Yim, D. Duff, and K. Roufas. Polybot: A modular reconfigurable robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 514–520, San Francisco, CA, USA, 2000.
19. Ying Zhang, Alex Golovinsky, Mark Yim, and Craig Eldershaw. An XML-based scripting language for chain-type modular robotic systems. In *Proceedings of the 8th Conference on Intelligent Autonomous Systems (IAS)*, 2004.

A Meta-model for Roles: Introducing Sessions

Valerio Genovese

Università di Torino, Dipartimento di Informatica
10149, Torino, Cso Svizzera 185, Italia
valerio.click@gmail.com

Abstract. Role is a widespread concept, it is used in many areas like MAS, DB, Programming Languages, Organizations, Security and OO modeling. Unfortunately, it seems that the literature is not actually able to give a uniform definition of roles, there exist several approaches that model roles in many different (and opposite) ways. Our aim is to build a formal framework through which we can describe different definitions appeared in the literature or implemented in computer systems. In particular we give a new role's foundation introducing *sessions*, which are a formal instrument to talk about role's states and we show how sessions may be useful to model relationships.

1 Introduction

The notion of role is a modelling concept strictly linked with interaction between entities. In natural language, we notice that terms like "student", "employee" or "president" are linked with a person who plays them and a *context* in which the *player* interact, the term "student" refers to a person that is a student in a specific university (eg. [1]). In a certain way, we can view roles as a pivotal concept to model an interaction, but problems arise because it is not completely clear how many different types of interactions exist and is possible to represent in the OO paradigm.

There are many definitions of roles, each one with a plausible approach based on intuition, practical needs and, sometimes, on a formal account. In security, roles are seen as a way to distribute permissions [2], in organizational models roles gives *powers* to their players in order to access an institution, in MAS roles could be seen as descriptions of the behavior which is expected by agents who play them [3], in ontology research roles are an anti-rigid notion founded on a player and a context [4], and many more. Even in the same field of research, there exist in the literature completely different notions of role which are in contrast with each other. Roles are not so easy to grasp, it seems that each different approach underlines a particular part of a common phenomenon not definable in a unique way.

The main goal of this work is to provide a flexible formal model for roles, which is able to catch the basic primitives behind the different role's accounts in

the literature, rather than a definition. If it is possible to define such a model, then we can study the key properties of roles in different implementations.

The paper is organized as follows. In section 2 we introduce the model. In section 3 we analyze in more depth sessions in relationships, showing links among states of different entities engaged in a collaboration. Conclusions close the paper.

2 A Logical Model for Roles

We define the formalism of the framework in a way as much general as possible, this gives us an unconstrained model where special constraints are added later in order to describe different approaches.

2.1 Universal Level

At the *universal* level we describe the relationship between natural and role types[1], in particular we define two relationship PL and RO through which we link roles to contexts and players (natural types) to roles.

Definition 1 An *universal model* is a tuple

$$< \mathsf{D}, \mathsf{Contexts}, \mathsf{Players}, \mathsf{Roles}, \mathsf{Attr}, \mathsf{Op}, \mathsf{Constraints}$$
$$\mathsf{PL}, \mathsf{RO}, \mathsf{AS}, \mathsf{OS}, \mathsf{RH}, \mathsf{PH}, \mathsf{CH} >$$

where:

- D is a domain of classes
- Contexts $\subseteq$ D is a set of institutions
- Players $\subseteq$ D is a set of potential players or *actors*
- Roles $\subset$ D is a finite set of *roles* $\{R_1, ..., R_n\}$
- Attr is a set of *attributes*
- Op is a set of *operations*
- Constraints is a set of *Constraints*

The static model has also a few relations:

- PL $\subseteq$ Players x Roles: this relation states, at the universal level, which are the players that can play a certain role.
- RO $\subseteq$ Roles x Contexts: each role is linked with one or more contexts.
- AS $\subseteq$ D x Attr: it is an attribute assignment relationship, through which we can assign to each class its attributes.
- OS $\subseteq$ D x Op: it is an operation assignment relationship, through which we can assign to each class its operations.
- RH $\subseteq$ Roles x Roles is a partial order relationship called role hierarchy, also written as $\geq_{RH}$. If $r <_{RH} r'$, we say that r inherits all Attr and Op which belong to r'.

[1] Natural types refer to the essence of the entities whereas role types depend on an accidental relationship to some other entity (context).

- PH $\subseteq$ Players x Players is a partial order relationship called player hierarchy, also written as $\geq_{PH}$. If $p <_{PH} p^{'}$, we say that p inherits all Attr and Op which belong to $p^{'}$.
- CH $\subseteq$ Contexts x Contexts is a partial order relationship called context hierarchy, also written as $\geq_{CH}$. Is $c <_{CH} c^{'}$, we say that c inherits from $c^{'}$.

Given the information contained in AS and OS relations, we use $\pi_{Attr}(o)$ and $\pi_{Op}(o)$ as shortcuts to refer about the set of attributes and operations defined for class o. At this point we can add into Constraints some logical rules in order to model different role notions. For example in powerJava each role type is linked with one and only one context type [5], so we can express this through the following constraint:

$$\forall x, y, z(x \in \mathsf{Roles}\ y, z \in \mathsf{Contexts}\ x\mathsf{RO}y \wedge x\mathsf{RO}z \rightarrow y = z)$$

2.2 Individual level

The *individual* part relies on the universal one and the elements of this level are individuals (or instances) of the types defined at the universal level.

Definition 2 A *snapshot model* is a tuple

$$< \mathsf{O}, \mathsf{I\_contexts}, \mathsf{I\_players}, \mathsf{I\_roles}, \mathsf{Sessions}, \mathsf{Val}, \mathsf{I\_contraints}$$
$$\mathsf{I}_{\mathsf{Roles}}, \mathsf{I\_Attributes}, \mathsf{I\_Operations}, \mathsf{I}_{\mathsf{Attr}} >$$

where:

- O is a *domain* of objects, for each object o is possible to refer to its attributes and operations through $\pi_{I\_Attr}(o)$ and $\pi_{I\_Op}(o)$, respectively.
- I_contexts $\subseteq$ O is a set of institutions which instantiate classes in Contexts.
- I_players $\subseteq$ O is a set of actors, which instantiate universals in Players.
- I_roles $\subset$ O is a set of *roles instances*.
- I_Attributes is the set of objects' attributes.
- I_Operations is the set of objects' operations.
- Sessions is a set of *sessions*, which keep the state of an interaction between actors and institutions (See Section 3).
- Val is a set of *values*.
- I_constraints is a set of integrity rules that constraint elements in the snapshot.

In this section we call elements in I_contexts, I_players and I_roles respectively, *institutions*, *actors* and *roles_instances*.
The snapshot model has also a few functions and relations:

- I_{Roles} is a *role assignment function* that assigns to each role R a relation on I_context x I_players x Sessions x I_roles.
- I_{Attr} is an *assignment function* which it takes as arguments an object $d \in$ O, and an attribute p $\in \pi_{I\_Attr}(d)$, if p has a value $v \in$ Val it returns it, $\emptyset$ otherwise.

When an object x is an individual of the universal y, we say that x instantiates y and, in order to express this in a formal way, we write $a :: b$ when a is an instance of b. In general if $x :: y$, attributes and operations defined for y at the universal level are assigned to x. If $a \in \pi_{Attr}(B)$ we write $\mathsf{x.a} \in \mathsf{I\_Attributes}$ as the attribute instance assigned to object x, the same holds for elements in $\mathsf{I\_Operations}$.

The role assignment function I_{Roles} gives us the notion of an actor who plays a role within a specific context: if $i :: x$ is an institution, $a :: y$ an actor, and $o :: R$ a role, $(i,a,o) \in I_{Roles}(R)$ is to be read as: "the object o represents agent a playing the role R in institution i". We will often write $R(i,a,o)$ for this statement, and we call o the *role instance*.

Suppose we have a role instance $\mathsf{employee}$, and that the value of the attribute salary is 1000 € usually, instead of writing $I_{Attr}(\mathsf{employee}, \mathsf{salary}) = \mathsf{1000}$, we write

$$\mathsf{salary(employee)} = \mathsf{1000}$$

The way we defined a snapshot leaves a lot of room for formulating further constraints in $\mathsf{I\_contraints}$ that may or may not be reasonable to assume, depending on the particular role's definition we have in mind. Here are a number:

1. *Dependence of roles on institutions.* In our model it is presupposed that the identity of a role instance depends not only on the role and the actor involved, but on an 'institution' as well. This is often, but not always, appropriate. We can mimic the case were the introduction on institutions is unnecessary with the introduction of a 'trivial' institution, and let $\mathsf{I\_contexts}$ contains only this trivial institution, as we do in section 3 when we model RBAC [2].
2. *Context coherence.* From an organizational point of view, there cannot be a student role's player without a teacher one, also it would not be sensible to talk about the context family without someone who plays the role of husband and another one being the wife. To express this constraint we can state, for example, the following integrity rule:

   $$\forall y \, (y :: Family \leftrightarrow \exists x, o, z, p \, (husband(y, x, o) \wedge wife(y, z, p))$$

   Which means that in the snapshot exists $y \in \mathsf{I\_contexts}$ if and only if there exist two role instances p and o which represent respectively an *husband* and a *wife* played by actors x and z in y.
3. *Complementary roles.* In general we can express the fact that playing a role R for an actor implies that there exists another actor playing a *complementary* role $R^{'}$ with the follwowing constraint:

   $$R(i, a, o) \leftrightarrow R^{'}(i, b, x)$$

2.3 The dynamic model

The dynamic model defines a structure to properly describe how the framework evolves as a consequence of executing an action on a snapshot. We well see in Section 4 and 5 how is up to this model to constraint agents' dynamics.

Definition 3 A *dynamic model* is a tuple

$$< \mathsf{S}, \mathsf{TM}, \mathsf{Actions}, \mathsf{Requirements}, \mathsf{D\_constraints}, \mathsf{I}_{\mathsf{Actions}}, \mathsf{I}_{\mathsf{Roles_t}}\ \pi_{\mathsf{Req}}, \mathsf{I}_{\mathsf{Requirements_t}} >$$

where:

- S is a set of *snapshots*.
- TM $\subseteq$ S x $\mathbb{N}$: it is a time assignment relationship, such that each snapshot has an associated unique time t. For the sake of simplicity we define a discrete time through positive natural numbers.
- Actions is a set of actions.
- Requirements is a set of requirements for playing roles in the dynamic model.
- D_contraints is a set of integrity rules that constraints the dynamic model.
- $\mathsf{I}_{\mathsf{Actions}}$ maps each action from Actions to a function on S. $I_{Actions}(s, a, t)$ tells us how the snapshot s changes as a result of executing action a at time t. This function returns a couple in TM that binds the resulting snapshot with time $t + 1$. In general, to express that at time t is carried action a we write a_t.
- About I_{Roles_t} we say that $R_t(i, a, o)$ is true if there exists, at a time t, the *role instance* $R(i, a, o)$.
- $\pi_{Req}(t, R)$ returns a subset of Requirements present at a given time t for the role of type R, which are the requirements that must be fulfilled in order to play roles of type R.

Intuitively, the snapshots in S represent the state of a system at a certain time. Looking at $\mathsf{I}_{\mathsf{Actions}}$ is possible to identify the *course* of actions as an ordered sequence of actions such that:

$$a_1; b_2; c_3$$

represents a system that evolves due to the execution of a,b and c at consecutive times. We refer to a particular snapshot using the time t as a reference, so that for instance π_{Attr_t} refers to π_{Attr} in the snapshot associated with t in TM.

We suppose that, for every time t, given an object p we can always say if it exist or not via the $exists_t$ operator, so that $exists_t(p)$ is true, if and only if p exists at time t, false otherwise. We write $exists(p)$ when p exists in all the snapshots of the dynamic model.

A particular aspect of the dynamic model is role *addition* and *deletion* model. It has actions corresponding to role assignment for each R, i and a, which are supposed to capture the effect of adding the role R within institution i to actor a, and other actions that represent the taking away from a the role R in institution i.

Of course, these actions will not be arbitrary. We first identify a number of minimal properties that the action of role assignment need to satisfy, then we describe a small set of possible actions that can be applied in the dynamic model.

Role Assignment

Let M be a snapshot.

$$\mathsf{M} = < \mathsf{O}, \mathsf{I\_contexts}, \mathsf{I\_players}, \mathsf{I\_roles}, \mathsf{Sessions}, \mathsf{Val}, \mathsf{I}_{\mathsf{Roles}}, \pi_{\mathsf{Attr}}, \pi_{\mathsf{Op}}, \mathsf{I}_{\mathsf{Attr}} >$$

Let $i \in \mathsf{I\_contexts}$, $a \in \mathsf{I\_players}$, and $R \in \mathsf{I\_Roles}$. There are two possibilities, if we want to assign role R to actor a: either if fails, or it succeeds. In the latter case, the resulting snapshot:

$$\mathsf{M}^{'} = < \mathsf{O}^{'}, \mathsf{I\_contexts}^{'}, \mathsf{I\_players}^{'}, \mathsf{I\_roles}^{'}, \mathsf{Sessions}^{'}, \mathsf{Val}^{'}, \mathsf{I}^{'}_{\mathsf{Roles}}, \pi^{'}_{\mathsf{Attr}}, \pi^{'}_{\mathsf{Op}}, \mathsf{I}^{'}_{\mathsf{Attr}} >$$

should satisfy the following properties:

- A role assignment may add at most one new object to the domain (namely the newly introduced qua-individual). $O^{'} = O \cup \{o\}$, where o may or may not already be in O.
- $\mathsf{I\_contexts}^{'} = \mathsf{I\_contexts}$ or $\mathsf{I\_contexts}^{'} = \mathsf{I\_contexts} \cup \{\mathsf{o}\}$.
- $\mathsf{I\_players}^{'} = \mathsf{I\_players}$ or $\mathsf{I\_players}^{'} = \mathsf{I\_players} \cup \{\mathsf{o}\}$.
- $\mathsf{I\_roles}^{'} = \mathsf{I\_roles}$, $\mathsf{Val}^{'} = \mathsf{Val}$. The sets of roles and possible values of attributes do not change.
- $I^{'}_{Roles}(R) = I_{Roles} \cup \{(i, a, o)\}$
- $\pi^{'}_{Attr}$ and $\pi^{'}_{Op}$ can be different if attributes and operations of a role are inherited by its player.
- $I^{'}_{Attr}$ is just like I_{Attr} with respect to the properties of objects different from i, a, and o.

For role addition and deletion actions we use, respectively $Req_t(i, a, R), R, i \hookrightarrow_t a$, and $Req_t(i, a, R), R, i \hookleftarrow_t a$. Then using the notation of dynamic logic we write:

$$[Req_t(i, a, R)?; R, i \hookrightarrow_t a]\phi$$

to express that, if actor a fills the requirements at time t ($Req_t(i, a, R)$ is True), after assigning role R within institution i at the same time t, ϕ is True in the resulting snapshot. If there are no particular Requirements (i.e. $\pi_{Req}(t, R) \in \emptyset$) we can omit Req_t. The above definition gives us the possibility to model that a role assignment introduces a role instance:

$$[R, i \hookrightarrow_t a]\exists x R(i, a, x)$$

or the fact that if a does not play the role R within institution i, then the role assignment introduces exactly one role instance:

$$(\neg\exists x R(i, a, x)) \rightarrow [R, i \hookrightarrow_t a]\exists! x R(i, a, x)$$

The dynamic level can be constrainted in order to model *inheritance of attributes and operations*, here we discuss only attributes, for operations the discussion is similar.

In the model, both roles and objects have properties. A natural constraint is that role-instances at least get all the properties that are defined for that role:

$$R_t(i, a, s, x) \rightarrow (\mathsf{attr} \in \pi_{Attr}(R) \rightarrow \exists v : \mathsf{attr}(\mathsf{x}) = v)$$

With respect to the question if the role-instance should 'inherit' all the properties of the original player object there are different possible answers.

For example, in powerJava [5], no such inheritance is assumed at all - the properties of the role instance are precisely those of the role, and we have that:

$$R_t(i, a, x) \rightarrow (\mathsf{attr} \in \pi_{Attr}(R) \leftrightarrow \exists v : \mathsf{attr}(\mathsf{x}) = v)$$

But other options are possible as well. For example, one alternative approach is that roles can be best seen as 'views' on a certain object, providing only a *subset* of the properties of the original object, like in Fibonacci [7]. A constraint which reflects that view is that the role-player has only the properties that are defined for the original object as well as for the role:

$$R_t(i, a, s) \rightarrow \pi_{Attr}(R) \subset \pi_{Attr_t}(a)$$

The opposite view is that roles *add* properties to the players. For example, in the Zope security model (like also in RBAC) we have the following:

$$[R, i \hookrightarrow_t a](\pi_{Attr_{t+1}}(a) = \pi_{Attr_t}(a) \cup_A \pi_{Attr}(R))$$

The same considerations hold for operations. In the above forumla we introduced an ad-hoc union operator $\cup_A$ that binds attributes of an object with attributes of a class instantiating them. For instance if we have an object o and a class T, the union $\pi_{Attr}(o) \cup_A \ \pi_{Attr}(T)$ add into $\mathsf{I\_Attributes}$ the elements o.a, a $\in$ $\pi_{Attr}(T)$.

Methods

There are other actions through which is possible to change the model as well, for instance objects may assign new values to their attributes [8]. Again, the effects of such changes may depend on choices made earlier (e.g. in the case of delegation, changing the attribute value of an object may change the value of that attribute also in some roles he plays).

3 Sessions and relationships

We explicitly introduce sessions because we argue that are strictly linked with the role's notion. As already said, we talk about sessions when is necessary to keep the state of an interaction between entities. Sessions in our model are a couple $(ID\,,\,K)$ where ID is an identifier and K a set of attributes and operations. If an attribute is in K it means that its value maintains a particular information

on the state of the interaction between an actor playing a role and an institution offering it. Operations in K are behavioral aspect of the interaction and they can change the value of the attributes that are in the same session, this means that operation in K can change attributes in K even if the are of different objects. For instance suppose to have *R(i,a,s,x)*, depending on what we want to model, we can look at sessions from three different points of view:

1. A session can collapse into one role instance [1,5,9] ($ID = x$). This means that attributes and operations in K are all a subset of $\pi_{Attr}(x) \cup \pi_{Op}(x)$ where $x \in$ I_roles.
2. A session can collapse into the actor [2,10] ($ID = a$). In that case peculiar attributes and operations for the interaction are linked with the object representing the actor.
3. A session can be an object with its own ID (like when we reify an association). It is important to underline that a session of this type can link different role instances embedding their attributes and operations in K, so that the state of a role instance a can be influenced by the behavior of another role individual b.

In powerJava the state of the interaction between a player and an institution is kept by the role instance:

$$R(i, a, s, x) \rightarrow \pi_K(s) \subseteq \pi_{Attr}(x) \cup \pi_{Op}(x)$$

Where $\pi_K(s)$ is a projection on the second element of couple s. The point is slightly different if roles are not instantiable, in this case we have:

$$R(i, a, s) \rightarrow \pi_K(s) \subseteq \pi_{Attr}(a) \cup \pi_{Op}(a)$$

The session notion gives the possibility to unify the state of the interaction between different roles instances or actors which participate in the same relationship or which are part of the same organizational model.

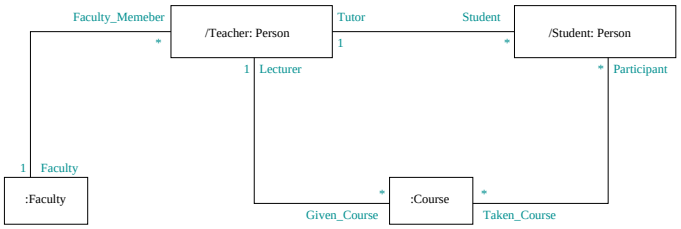


Fig. 1. UML collaboration diagram

In UML, roles serve two purposes: they label association ends, and they act as type specifiers in the scope of a collaborations (so-called classifier roles) [10].

In Figure 1 the labels of the association ends correspond to our *roles*, a straight line between a Teacher and a Student identify an interaction between them, where tutor and student are the roles through which the interaction takes place.

Depending on what we have in mind, we can express the interaction between two instances of Person (one acting as a Teacher and the other one as Student) in two different ways, if $x :: Person$, $y :: Person$, $tutor :: Tutor$ and $student :: Student$[2] we have:

1. $Tutor(y, x, q, tutor) \wedge Student(x, y, q^{'}, student)$
2. $Tutor(y, x, q, tutor) \wedge Student(x, y, q, student)$

Notice that x and y are both in I_contexts and I_players, because they offer and play roles at the same time. In the first view we have two separate sessions each one representing a specific direction of the association between x and y, whereas in the second approach a common session q unifies the two-way association seeing it as a unique interaction with a unique state for both directions ($x \rightarrow y$ and $y \rightarrow x$). It must be said that is not mandatory to model the interaction between x and y with *role instances*, if we do not want roles to be instantiated we simply let sessions refer to attributes and operations of x and y.

The UML collaboration diagram (Figure 1) defines, at the specification level, how instances of different classes must behave in order to be engaged in the collaboration in a sort of relationship's pattern. In Figure 2 we represent role instances inside the context that offers them, the relation of playing a role is translated through an arrow which goes from the actor to the role instance played.

The view of putting the role tutor inside the object studentD, together with having all objects being at the same time contexts and players of some roles, could seem counter intuitive, but is extremely powerful. Role instances are seen as set of *affordances* [11] that let the actor interact with another entity, in general an actor plays a role which is linked with a context, and the fact of playing that role gives him the *power* to modify the properties of the context. With this example in mind we can now translate the diagram in Figure 1 representing it through a set of constraints at the individual level [3]:

$$Tutor(student, teacher, s_1, tutor) \wedge$$
$$Faculty\_Member(faculty, teacher, s_2, faculty\_member) \wedge$$
$$Lecturer(course, teacher, s_3, lecturer) \wedge Student(teacher, student, s_1, student) \wedge$$
$$Participant(course, student, s_4, participant) \wedge$$
$$Faculty(teacher, faculty, s_2, faculty) \wedge$$
$$Taken\_Course(student, course, s_4, taken\_course) \wedge$$
$$Given\_Course(teacher, course, s_3, given\_course)$$

[2] In this section we refer to classes with the first letter capitalized in order to distinguish them from instances which are in lower case.

[3] In order to avoid confusion we refer to $teacher$, $student$, $course$,and $faculty$ as instances of the classes involved in the collaboration diagram.

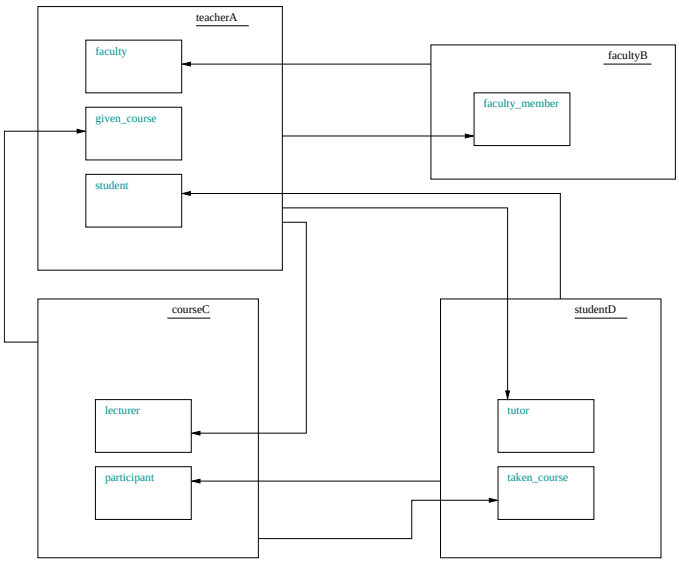


Fig. 2. Example of object diagram with roles

This predicate represents a set of constraints that have to be applied to all entities that want to be engaged in the collaboration diagram in Figure 1. So it is impossible to play the role lecturer without offering the role student, and without being engaged in all others associations implied in the collaboration diagram.

We said that playing a role always translates into modeling an interaction, and that the state and behavior of this interaction is kept by a subset of *attributes* and *operations* of the entity engaged in the relationship. We introduce the term *session* to refer to this subset of elements because this abstraction let us model, in a formal and hopefully clear way, the links that relate the *states* of the elements playing roles in a relationships.

In general, when attributes' values in a session s_1 are influenced by operations or actions carried out by other roles which elements are in another session s_2, we need to express an integrity rule that links the states of s_1 and s_2.

Referring to Figure 2, suppose that *faculty_member* and *tutor* have an attribute *num_courses* which value counts the number of courses held by the *teacherA*, if *teacherA* stops playing *lecturer* in *courseC*, *num_courses* in both *faculty_member* and *tutor* should be decreased by one. There could also be a case where an action carried out as *tutor* can modify *lecturer*'s state (i.e the execution of a *tutor*'s *operation* can change one or more lecturer's *attributes*).

Then we can define the following integrity rule in D_contraints of the dynamic model:

$$\forall z, p, q \ : \\ p :: Faculty \ \wedge \ q :: Student \ \wedge z :: Teacher \wedge \\ faculty\_member_t(p, z, s_1, x) \ \wedge \ tutor_t(q, z, s_2, y) \\ \rightarrow \\ num\_courses(x) = num\_courses(y) = \beta$$

Where β is the number of lecturer instances played by z. Notice that in the dynamic model the value of β can always be deduced analyzing the set of snapshots in S.

With the introduction of sessions we argue that to model properly a relationship is important to talk about states that are strictly linked with the role played, and that roles can not be simple labels of association ends.

4 Conclusions

In this article we extend and improve the framework for modelling roles introduced in [8], moreover we redefine the session notion in order to analyze in more depth how roles are linked with relationships. In [8] it has been shown that the model is able to grasp different role notions, the aim is to find the basic elements which can describe what roles are. We think that relationships are the right place to investigate and to find a foundation of roles.

References

1. Frank Loebe: Abstract vs. social roles - a refined top-level ontological analysis. In Procs. of AAAI Fall Symposium on Roles, an interdisciplinary perspective Series Technical Reports FS-05-08. pages 93-100 (2005)
2. Ravi S.Sandhu, Edkward J.Coyne: Role-Based Access Control Models. IEEE Computer, Volume 29, Number 2 38-47 (1996)
3. L. van der Torre, Guido Boella: Organizations as socially constructed agents in the agent oriented paradigm. In Procs. of ESAW'04 (2004)
4. Claudio Masolo, Laure Vieu, Emanuele Bottazzi, Carola Catenacci, Roberta Ferrario, Aldo Gangemi, Nicola Guarino: Social roles and their descriptions. In Procs. KR2004, Whistler, Canada, June 2-5, 2004, pp.267-277 (2004)
5. Matteo Baldoni, L. van der Torre, Guido Boella: Interaction between objects in powerJava. Journal of Object Technology(JOT) (2006)
6. M. Baldoni, C. Baroglio, A. Martelli, V. Patti: Reasoning about interaction protocols for customizing web service selection and composition. Journal of LOgic and Algebraic Programming, special issue on Web Services and Formal Methods,70(1):53-73 (2007)
7. A.Albano, R.Begamini, G.Ghelli, R. Orsini: An object data model with roles. In R. Agrawal, S. Baker, and D. Bell, editors, Proc. of the Nineteenth Intl. Conf. on Very Large Data Bases (VLDB), Dublin, Ireland, pages 39-51, San Mateo, CA, 1993. Morgan Kaufmann (1993)
8. Valerio Genovese: Towards a general framework for modelling roles. In Guido Boella, Leon van der Torre, Harko Verhagen, eds.: Normative Multi-agent Systems. Number 07122 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2007)

9. Stephan Herrmann: Programming with Roles in ObjectTeams/Java. In proc. AAAI Fall Symposium 2005: Roles, an iterdisciplinary perspective (2005)
10. Friedrich Steimann: On the representation of roles in object-oriented and conceptual modelling. Data and Knowledge Engineering, 35:83-848 (2000)
11. Matteo Baldoni, Guido Boella, L. van der Torre: Modelling the interaction between objects: Roles as affordances. In Procs. Knowledhe Science, Engineering and Management, First International Conference, KSEM 2006, Guilin, China, August 5-8, volume 4092 of Lecture Notes in Computer Science, pages 42-54. Springer, 2006 (2006)

Role Representation Model Using OWL and SWRL

Kouji Kozaki, Eiichi Sunagawa, Yoshinobu Kitamura, Riichiro Mizoguchi

The Institute of Scientific and Industrial Research (ISIR), Osaka University
8-1 Mihogaoka, Ibaraki, Osaka, 567-0047 Japan
{sunagawa, kozaki, kita, miz}@ei.sanken.osaka-u.ac.jp

Abstract. Role is very important in ontology engineering. Although OWL has been available for ontology representation, consideration about roles is not enough. It can cause to decrease semantic interoperability of ontologies because of conceptual gaps between OWL and developers. To overcome this difficulty, this paper presents some consideration for dealing with roles using OWL.

1. Introduction

Ontology is one of the key technologies for realization of the Semantic Web. To represent web-ontologies, OWL and SWRL have been published as a W3C Recommendation. Although there are many tools for ontology development in OWL and SWRL, few of them provide a higher-level framework for conceptualization of the target world with fundamental discussion. That can cause to decrease semantic interoperability of ontologies because developers need to devise idiosyncratic patterns for building their own ontologies for themselves and such patterns will lack compatibility with others. In this research, we focus on roles [1, 2, 3, 5, 7] as one of the common and typical semantic primitives in ontology development, and investigate representation model for dealing with characteristics of roles in OWL and SWRL justified by fundamental consideration. It contributes to increasing semantic interoperability of roles by providing an infrastructure for role representation.

This paper is organized as follows. Section 2 clarifies roles treated in this paper and summarizes characteristics of roles as requirements for representation model. Section 3 evaluates some examples of role representation. After that, role representation model in this research is presented. Section 4 describes some related work. And, Section 5 concludes this paper with description about some future work.

2. Characteristics of Roles (Requirements)

2.1 Roles in Our Model

In this section, we summarize fundamental schema of our role model proposed in previous work [10]. The fundamental scheme of roles at the instance level is the following (see the lower diagram in Fig. 1): *"In Osaka high school, John plays*

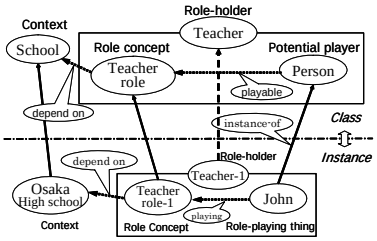


Fig. 1. Fundamental schema of a role concept and role holder

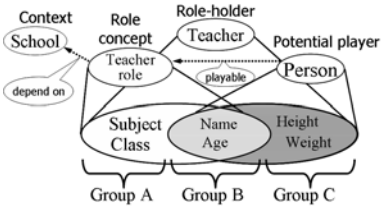


Fig. 2. Conceptual framework of a role, player and role-holder

teacher role-1 and thereby becomes teacher-1". This can be generalized to the class level (see the upper diagram in Fig. 1): *"In schools, there are **persons** who play **teacher roles**[1] and thereby become **teachers**"*. By play, we mean "act as", that is, it contingently acts as the role (role concept). By "**teacher**", we mean a class of persons who are playing teacher roles.

We introduced a couple of important concepts to enable finer distinction among role-related concepts: **Role concept**, **Role holder**, **Potential player** and **Role-playing thing**. In the above example, these terms are used as *"In a context, there are **potential players** who can play **role concepts** and thereby become **role-holders**."* By **context**, we mean a class of things to be considered as a whole. It includes entities and relations. R**ole concept** is defined as a class of concepts which are played by something within a context. By **potential player**, we mean a class of things which are able to play an instance of a role concept. In many cases[2], **basic concepts** (natural types) can be used as potential player class. In this example, we say a person can play an instance of *teacher role*. And, when a person (an instance of person class) is actually playing a *teacher role*, he/she thereby becomes an individual *teacher **role-holder***. This means the conventional concept, **player**, is divided into two: One is **potential player** (a role-playable thing) which at the class level, means a class of entities which can play a role of interest and the other is a **role-playing thing**, which is an entity playing the role at the instance level. At the same time, the conventional player link is divided into two kinds: one is playable link (class level) and the other is playing link (instance level). Role holder class is an abstraction of a composition of role-playing thing and an instance of role concept, as is shown in Figs. 1 and 2.

Fig.2. shows the conceptual framework of role model we have proposed. There are two kinds of properties: those of teacher role and person. All of the properties are divided into three groups. Properties of group A are determined by the definition of the role concept itself independently of its player. The second group B is shared by both of the role concept and the potential player. And, the last group C is what the role concept does not care about. Generally, a role concept is defined by describing its properties of group A together with some from group B which are shared by a potential player but come originally from the role concept. Its potential player class is defined by itself context-independently and is used as a constraint of the potential player of the role concept. And, the role-holder is defined as a result of the above two definition operations and eventually includes all of three kinds of properties. Therefore, the individual corresponding to a teacher role holder is the compound of these two instances and totally dependent on them.

---

[1] When we mention a particular role, we put "role" after its name.

[2] In some case, role holders can be used as potential player class. (see section 2.2 (5))

2.2 Requirements for Role Representation Model

In the following items, we summarize characteristics of role concepts. They are referred to as criteria for evaluating role representation models in the next sections.

***(1) Context dependency*:** Context dependence is critical to describe that roles cannot be determined without their contexts and entities change their roles to play according to changes of their contexts.

***(2) Identity of Role concepts*:** Identities of roles is needed to discuss whether two roles are the same or not. For example, when a person is reinstated in his former position, is the role he/she is playing now the same as the one he/she has played? Identities of the roles may answer this question. Furthermore, it enables to represent a vacant post by an individual of a role which is left un-played.

***(3) Distinction between role concepts and role-holders*:** The distinction between role concepts and role-holders is represented the conceptual model discussed in section 2.1. It solves counting problem described in (11).

***(4) Part-whole relation associated with roles and players*:** An object, which is recognized as a whole thing for its part thing(s), can be conceptualized from at least two aspects. From one aspect, the whole thing consists of constituents which build up it (e.g. wheels of bicycle). From the other, it has a conceptual structure which determines roles played by the constituents (e.g. a steering wheel and driving wheel of bicycle). Hence, in some cases, the whole thing is described as a composition of the role-playing thing(s) and a set of the roles. This is similar to a description of a crystal as a composition of "constituents" in the crystal structure and the "crystal structure" without the constituents. By a context for roles in this paper, we mean a conceptual structure. It corresponds roughly to particular patterns of relationships connected with roles by Sowa [9], associations or collaborations in UML.

***(5) Compound role concepts*:** Some role needs to be played together with other roles. And, in some case, a player stops playing one of the roles, and then, some of others will automatically be un-played according to interdependency. Such a relation between roles is discussed in other researches as "requirement" [7] and "roles can play roles" [8]. In our terminology, "role-holders can play roles". Such a role concept depends on multiple contexts. For example, teacher can be recognized not only as a staff member of a school but also as a person who teaches students. So, teacher role is interpreted as a compound of school staff role and teaching agent role. So, here, we can identify two kinds of roles according to the complexity of their context dependencies: primitive role concepts and compound role concepts. The former has a single context-dependency and latter has multiple context dependency. A school staff role in school context and a teaching agent role in teaching action context are primitive role. A teacher role is a compound role of them. In order to deal with compound role concepts, here we introduce **Role Aggregation**, which is based on decomposition of the compound role and determination of essential context for it [10].

***(6) An individual plays multiple roles*:** An individual can play multiple roles at the same time. For example, an instance of *Person* may play a *Teacher Role* and a *Husband Role* at the same time.

***(7) Individuals of role concepts*:** Individuals of role concepts have the following two characteristics. (a) They cannot exist if individuals of their contexts do not exist

because roles are externally founded [3,7]. (b) Because roles are dynamic [7], they have two states: played and un-played. (c) They have their own identities independently of their states and their players and are regarded as defective instances until played by some individuals.

***(8) Individuals of role-holders*:** An individual of a role-holder is composed of individuals of a role concept and its player. The identity (ID) of the individual of the role-holder is a function of the IDs of the role concept (ID_{Role}) and of the player (ID_{Player}). That is, $ID_{Role\text{-}Holder}=f$ (ID_{Role}, ID_{Player}) in which both arguments are mandatory for $ID_{RoleHolder}$, in which "*f*" is bijective (surjective and injective)

***(9) Disappearance of individuals of role-holder*s:** In connection with (8) and (9), individuals of role-holders disappear when (a) its player disappears, (b) its role disappears and its player quits playing the role.

***(10) Solution of counting problem*:** For example, the number of passengers taking a certain means of transportation in one week may be greater than the number of individual persons traveling with that means during the same period [4,11]. This problem, so-called Counting Problem, can be solved by separately counting identities of individuals (players) and that of Passenger role-holders which are recognized every time that the players play Passenger role. For example, when we need to count the number of passengers, we use the $ID_{Role\ holder}$, and when we need to count the number of persons, we use ID_{Player} instead of $ID_{Role\text{-}Holder}$.

***(11) Players of compound role concepts*:** Individuals of roles as constituents of a compound role need to be played together by the same individual.

3. Representation of Roles Using OWL and SWRL

3.1. Examples of Role Representation

Fig.3 shows three examples of role representation model in OWL. In this section, we evaluate each of them according to the requirements for dealing with characteristics of roles discussed in Section 2.2. Here, we refer a Teacher Role, which depends on a School as its context, exampled in Section 2.1. Table 1 shows the result of comparison among the examples. (d) and (d)+ in the table shows representation ability of the proposed models explained in the next section.

Example 1 (in Fig.3-a): A role as a Teacher is dealt with in *teacherOf* property. This property may represent the role which is determined in e.g. "teacher-student relation". In this model, however, context dependency of role concepts is implicit (see (1) in the former section). That causes a critical problem because the context dependency relates essentially to other characteristics.

Example 2 (in Fig.3-b): This model represents a context of Teacher explicitly. However, the role is dealt with still in a property. That can complicate management of identity of roles in its instance model (see (2)). For example, it is difficult to describe that, after some person quits his/her job as a Teacher, other person fills the same post as the Teacher. Moreover, this model can not represent state of the role concept: played or un-played (see (7)-b). It means, for example, a vacant post can hardly be represented and identified.

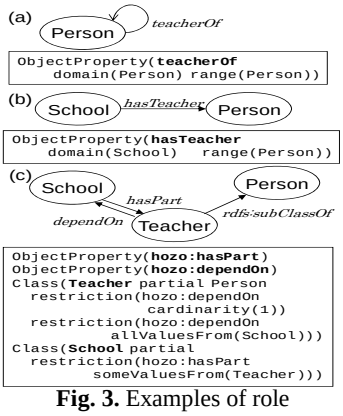


Fig. 3. Examples of role representation models

Table 1. Comparison of role representation models

Characteristics of Role Concepts and Role-Holders	*(a)*	*(b)*	*(c)*	*(d)*	*(d)+*
(1) Context dependency	-	OK	OK	OK	OK
(2) Identity of Role concept	-	-	OK	OK	OK
(3) Distinction between role concepts and role-holders	-	-	-	-/OK	OK
(4) Part-whole relation associated with roles and players	-	-/OK	-/OK	-/OK	OK
(5) Compound role concepts	-/OK	-/OK	-/OK	OK	OK
(6) An individual plays multiple roles	OK	OK	OK	OK	OK
(7) Individuals of role concepts	-	-	-	-/OK	OK
(8) Individuals of role-holders	-	-	OK	-/OK	OK
(9) Disappearance of individuals of role-holders	OK	OK	-/OK	-/OK	OK
(10) Solution of counting problem	-	-	-	OK	OK
(11) Players of compound role concepts	-/OK	-/OK	-/OK	-	OK

OK:represented, -:not represented, -/OK:represented partially

Example 3 (in Fig.3-c): The *hasPart* property in this model means School consists of Teacher(s) here. And a restriction on *dependOn* property in Teacher class expresses that a Teacher depends on School as its context. This model is superior to the above two models because their problems can be solved in this model. However, a Teacher is classified into a Person in confusion between role concepts, role-holders and basic concepts (see (3)). Hence, according to the semantics of *rdfs:subClassOf*, an instance of a Teacher and its player (an instance of Person) are required to be one and the same instance. That causes the player cannot stop to be an instance of a Teacher without stopping to be an instance of a Person, i.e., deletion of an instance of a Teacher brings with deletion of an instance of a Person (see (9)). Furthermore, in this model, the Counting Problem cannot be solved because it is necessary for solution of the problem to distinct role-holders from role concepts (see (3) and (10)).

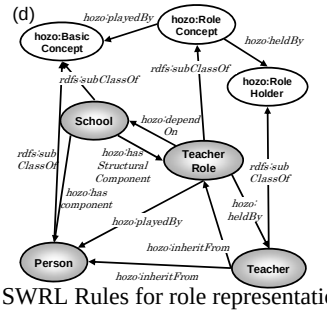
3.2. Role Representation Model

In Fig. 4(d), we represent our role model in OWL for dealing with characteristics of roles with fundamental consideration as faithfully as possible. We define some properties and classes which are indicated by namespace “hozo:”. For example, hozo:BasicConcept class, hozo:RoleConcept class and hozo:RoleHolder class express basic concepts, role concepts and role-holders respectively. hozo:playedBy property represents a relation between classes of role concept and classes of potential player. This property indicates role-playable thing discussed in 2.1. When a relation between an instance of role concept and player is represented as hozo:playedBy property, the property means a playing relation between them. hozo:RoleHolder class represents a role holder.

Fig. 4(d)+ gives rules which are applied into classes and properties in this role representation. Theey are written in a human-readable style and can be implemented in SWRL rules like an example under the table. They are not only applied to instance models for inference, but also implying our policies on using the classes and properties in this section for describing characteristics of roles. For example, Rule-03 and 04 require that we describe a role concept with two properties (*hozo:dependOn* and *hozo:hasStructuralComponent*) among the role and a class as its context.

In the following items, we evaluate this model also with reference to characteristics of roles summarized above. The evaluations are shown in Table 1 as (d) and (d+). In

(d)



(d)+ : SWRL Rules for role representation model

	Antecedent	Consequent
Rule-01	hozo:playedBy(?x, ?y) ∧ hozo:hasStructuralComponent(?z, ?x)	hozo:hasComponent(?z, ?y)
Rule-02	hozo:playedBy(?x, ?y) ∧ hozo:hasComponent(?z, ?y)	hozo:hasComponent(?z, ?x)
Rule-03	hozo:structuralComponent(?x, ?y)	hozo:dependOn (?y, ?x)
Rule-04	hozo:dependOn (?x, ?y)	hozo:structuralComponent(?y, ?x)
Rule-05	hozo:playedBy(?x, ?y)	hozo:heldBy(?x, ?z)
Rule-06	hozo:heldBy(?x, ?y)	hozo:playedBy(?y, ?x)
Rule-07	hozo:playedBy(?x, ?y) ∧ hozo:heldBy(?x, ?z)	hozo:inheritFrom(?z, ?x)
Rule-08	hozo:inheritFrom(?x, ?y) ∧ hozo:RoleConcept(?y)	hozo:heldby(?y, ?x)
Rule-09	hozo:aggregateOf(?x, ?y) ∧ hozo:playedBy(?x, ?z)	hozo:playedBy(?y, ?z)
Rule-10	hozo:aggregateOf(?x, ?y) ∧ hozo:playedBy(?y, ?z)	hozo:playedBy(?x,?z)
Rule-11	aggregateOf(?x, ?y)	differentFrom(?x, ?y)

Example : Rule-01
Implies(Antecedent(hozo:playedBy(I-variable(x) I-variable(y))
hozo:hasStructuralComponent(I-variable(z) I-variable(x)))
Consequent(hozo:hasComponent(I-variable(z)) I-variable(y)))

```
ObjectProperty(hozo:dependOn
    domain(hozo:RoleConcept))
ObjectProperty(hozo:playedBy
    domain(hozo:RoleConcept)
    range(hozo:BasicConcept))
ObjectProperty(hozo:inheritFrom
    domain(hozo:RoleHolder))
ObjectProperty(hozo:hasComponent
    range(hozo:BasicConcept))
ObjectProperty(hozo:hasStructuralComponent
    range(hozo:RoleConcept))
Class(hozo:BasicConcept partial
    DisjointClasses(hozo:RoleConcept hozo:RoleHolder))
Class(hozo:RoleConcept partial
    restriction(hozo:dependOn cardinality(1))
    restriction(hozo:playedBy maxCardinality(1)))
     restriction(hozo:heldBy maxCardinality(1)))
Class(hozo:RoleHolder partial
    restriction(hozo:inheritFrom cardinality(2))
    restriction(hozo:inheritFrom
                someValuesFrom(hozo:RoleConcept))
    restriction(hozo:inheritFrom
                someValuesFrom(hozo:BasicConcept)))
Class(TeacherRole partial hozo:RoleConcept
    restriction(hozo:dependOn allValuesFrom(School))
    restriction(hozo:playedBy allValuesFrom(person)))
Class(Teacher partial hozo:RoleHolder
    restriction(hozo:inheritFrom
                someValuesFrom(TeacherRole))
    restriction(hozo:inheritFrom
                someValuesFrom(Person)))
Class(School partial
    restriction(hozo:hasStructuralComponent
                someValuesFrom(TeacherRole))
    restriction(hozo:hasComponent
                someValuesFrom(Person)))
```

Fig. 4. Role represntation model in Hozo

column (d), the model is evaluated only within the description of this model in OWL. And, in column (d+), it is done within the descriptions in OWL and SWRL.

***(1)Context dependency*:** The definition of hozo:RoleConcept has a restriction on this property to have exactly one *hozo:dependOn* property. It represents all role concepts depend on another class as their context.

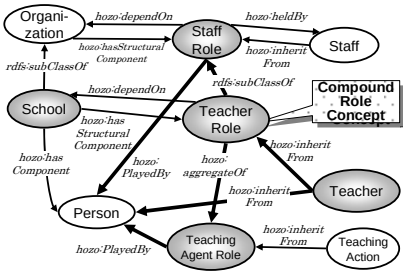
***(2) Identity of Role concepts*:** Roles are conceptualized as classes and are categorized into hozo:RoleConcept as its subclasses. One of the major contributions by treating roles as not properties but classes in the syntax of OWL is to make management of identities of roles easier.

(3) Distinction between role concepts and role-holders: In the same way as roles, role-holders are categorized into hozo:RoleHolder. And, they are discriminated from each other explicitly. Role-Holders are described with *hozo:inheritFrom*. This property is used for representing that a role-holder inherits definitions from role concept or its player. But the property does not imply inheritance of identity, and in that respect *hozo:inheritFrom* differs from *rdfs:subClassOf*.

***(4) Part-whole relation associated with roles and players*:** Part-whole relation is represented by two properties (*hozo:hasStructuralComponent* and *hozo:hasComponent*) and rules for reasoning on them (Rule-01,02 in Table 2).

***(5) Compound role concepts*:** Fig. 5 shows an extended model of the one in Fig. 4 for role aggregation. In this example, a Teacher Role is a compound role concept defined by aggregation of a Staff Role and a Teaching Agent Role. A compound role concept is described by role aggregation using two properties: *rdfs:subClassOf* and *hozo:aggregateOf*. The latter means that, a role concept in its range inherits some properties from one in its domain.

***(6) An individual plays multiple roles*:** This can be represented by several instances of role concepts are connected with one and the same individual through *hozo:playedBy*.




```
ObjectProperty(hozo:aggregateOf
    domain(hozo:RoleConcept)
    range(hozo:RoleConcept))
Class(Teaching Agent Role partial hozo:RoleConcept
    restriction(hozo:playedBy
                allValuesFrom(Person)))
Class(Teacher Role partial Staff Role
    restriction(hozo:dependOn
                allValuesFrom(School))
    restriction(hozo:playedBy
                allValuesFrom(Person))
    restriction(hozo:aggregateOf
                allValuesFrom(
                     Teaching Agent Role)))
```

Fig. 5. Representation model for Role Aggregation in Hozo

***(7) Individuals of role concepts*:** (a) By a restriction on cardinalities of *hozo:dependOn* in hozo:RoleConcept and Rule-03 and 04 in Table 2, it is described that an individual of a role concept exists always accompany with an instance of its context. (b) Two states of a role: played or un-played are distinguished by whether the role has *hozo:playedBy* or not. (c) Individuals of role concepts are identified as instances of hozo:RoleConcept.

***(8) Individuals of role-holders*:** An instances of a role holder exists iff *hozo:playedBy* holds between an instance of its role concept and one of its player. And, the instance of role-holder inherits the definitions from instances of the role concept and the player by *hozo:inheritFrom* (Rule-05~09 in Table 2).

***(9) Disappearance of individuals of role-holders*:** Restrictions on properties of hozo:RoleHolder and Rule-05~09 described in (8) mean also that iff their onditions are not fulfilled, an individuals of role-holder cannot exist.

***(10) Solution of counting problem*:** An instance of a player and one of a role-holder are distinguished by their own identities. Counting Problem can be solved by counting them separately.

***(11) Players of compound role concepts*:** Rule-10~12 in Table 2 describe that a player of a compound role must also play other roles connected with the compound role by *hozo:aggregateOf* simultaneously.

4. Related Work

W3C has started up Semantic Web Best Practice and Deployment Working Group for providing typical semantic primitives of ontologies as Ontology Engineering Patterns. For example, in the draft on simple part-whole relations[3], it is described that "It is important to realize that making, e.g. Engine a subclass of e.g. CarPart means that all engines are car parts - which is simply not true". These problems show exactly why we need to discriminate roles from the others for development of ontology. In the role representation model presented in this paper, CarPart and Engine are regarded respectively as a role concept and as a player of the role. In this way, the role representation model contributes to assure semantic interoperability of roles.

Guarino and his colleagues aim to establish a formal framework for dealing with roles [3, 7]. And Gangemi and Mika introduce an ontology for representing a context and states of affairs, called D&S, and its application to roles [2]. Our notions of role

[3] http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/

concepts share a lot with their theory of roles; that is, context-dependence, specialization of roles, and so on. But our role model differs from their work on other two points. Firstly, we focus on context-dependence of a role concept. So, time dependence of a role concept is treated implicitly in our framework because an entity changes its roles to play according to its aspect without time passing. Secondly, we distinguish role concepts and role holders.

5. Conclusion and Future Work

In this paper, we have discussed characteristics of roles and developed a role representation model using OWL and SWRL. The model covers major important aspects of roles and is available to represent role and their characteristics. It not only contributes to semantic interoperability of ontologies but also affords clues for solving problems caused by confusion of roles, the counting problem and so on.

As future work, we plan to investigate some other characteristics of roles such as instance management of roles (when it is created or deleted?), categories of roles, and so on. The instance management is the most serious among them. In order to clearly understand playable, playing, depend-on relations, we need to investigate when and how the related instances appear and disappear in what interdependence.

References

1. Fan, J., Barker, K.,et al.: Representing Roles and Purpose. In Proc. of the International Conference on Knowledge Capture (K-Cap2001), pp.38–43, Victoria, B.C., Canada (2001)
2. Gangemi, A., Mika, P.: Understanding the Semantic Web through Descriptions and Situations, ODBASE 2003, Catania, Italy (2003)
3. Guarino, N.: Concepts, attributes and arbitrary relations, Data & Knowledge Engineering 8 (1992) 249-2615.
4. Guizzardi, G. “Agent Roles, Qua Individuals and The Counting Problem”, in Software Engineering of Multi-Agent Systems, vol. IV, Springer-Verlag, 2006.
5. Guizzardi, G., “Ontological Foundations for Structural Conceptual Models”, Telematica Instituut Fundamental Research Series, Universal Press, The Netherlands, 2005
6. Loebe, F.: Abstract vs. Social Roles - A Refined Top-Level Ontological Analysis. Papers from the AAAI Fall Symposium Technical Report FS-05-08, pp.93-100, USA (2005)
7. Masolo, C., Vieu, L., Bottazzi, E., et al.: Social Roles and their Descriptions. In Proc. of the 9th International Conference on the Principles of Knowledge Representation and Reasoning (KR2004), pp.267–277, Whistler, Canada (2004)
8. Steimann, F.: On the Representation of Roles in Object-oriented and Conceptual Modeling, Data & Knowledge Engineering 35 (2000) 83-106
9. Sowa, J. F. 1988. Using a lexicon of canonical graphs in a semantic interpreter, in Relational models of the lexicon (1988)
10. Sunagawa, E., Kozaki, K., Kitamura, Y., Mizoguchi, R.: A Framework for Organizing Role Concepts in Ontology Development Tool: Hozo, Papers from the AAAI Fall Symposium Technical Report FS-05-08, pp. 136-143, USA (2005)
11. Wieringa, R. J., Jonge W. de, Spruit P. A.: Using Dynamic Classes and Role Classes to Model Object Migration. Theory and Practice of Object Systems 1 (1995) 61-83

Towards a Definition of Roles for Software Engineering and Programming Languages

Frank Loebe

Research Group Ontologies in Medicine (Onto-Med)
Institute of Medical Informatics, Statistics and Epidemiology (IMISE)
University of Leipzig, Germany
frank.loebe@imise.uni-leipzig.de
http://www.onto-med.de

1 An Analytic, Ontology-oriented View on Roles

Analyzing the notion of role in the literature yields a plurality of views and definitions. In [1–3], we study a broad range of approaches in order to interrelate and harmonize them (where possible) in the context of an ontological framework, whose central component is the top-level ontology General Formal Ontology (GFO)[1] [4].

A major goal of our work is the provision of a role definition which maximizes the coverage of applications of the term "role". To the extent possible this should be independent from specific application areas, spanning from conceptual modeling to software engineering to linguistics, etc. This leads to a very general, yet weak, analytical definition for the notion of role:

Definition 1 A *role* is an entity which is dependent on two other entities, referred to as the *player* of the role and the *context* of the role.

For instance, a student role requires an individual human being as a player, and the role arises in a social context, e.g. as established by a particular university. In general, entities recognizable as players or contexts are hardly constrained in our approach. However, the dependence between roles and contexts frequently involves kinds of part-of relationships.

To render these issues more precisely, a major aspect of our approach must be noted: we distinguish role individuals and role classes.[2] Given this distinction, role individuals satisfy Definition 1 literally. Player and role individuals are distinct entities in our approach, and players are characterizable independently of the role and its context, by means of other kinds of classes, often called *natural types* or *object types*. On the class level, the notion of player class is thus merely of indirect relevance, because role individuals require player individuals. However, context classes are strongly intertwined with role classes, where we advocate that the two exhibit definitional interdependences, cf. also [5].

[1] http://www.onto-med.de/ontologies/gfo/

[2] We use role classes here to align with object-oriented terminology. In [1–3], role classes are called role universals.

Furthermore, analyses of the ontological categories of typical role contexts result in different role kinds. According to those categories identified – relations, processes, and social systems – we distinguish *relational*, *processual* and *social roles*. Relational and processual roles are unified into *abstract roles*, because especially role classes of these two kinds are defined primarily *externally*, with respect to certain contexts, as discussed in [1, sect. 2.3]. In contrast, the context of social roles remains often vague, whereas *internal* characteristics like typical behavior, properties, and relations form important aspects exhibited by social roles. Moreover, in connection with social roles, dynamicity (e.g. that entities may gain or lose roles) and anti-rigidity are two important notions [5, 6], which we understand as referring to interrelations of role classes and natural types.

2 Towards a Definition for Software Engineering and Programming Languages

Theories of programming and software engineering, in different shapes like object-, agent-, or aspect-orientation, form a major area of using and applying roles in computer science. In this context it does not appear reasonable to argue for the direct adoption of Definition 1 above, which would be too weak while the broadness of coverage is not necessary. Most occurrences of roles in this area seem to require properties for roles and involvement in complex systems, closely resembling social roles from above (a slight generalization to non-social objects may be required). For a common use of "role" in this area we propose the following adaptation of Definition 1 as a working hypothesis:[3]

Definition 2 A role R is an entity which mediates between a context C, comprehended as a system or a society of interrelated entities $E_1, E_2, \ldots$, and exactly one of these entities, E_i. R depends on both, E_i and C, and it exhibits specific properties and behavior (which for *pure roles* are based *exclusively* on the context C).

Note that this definition exhibits some similarity to the UML 2.0 definition of role [7, p. 575]. Several further strengthened definitions may be developed for specific fields, by incorporating roles into their domain vocabulary which then allows one to express a number of additional features. For instance, in object-oriented approaches one may augment Definition 2 with features of the interconnections between (type) objects and role objects, e.g. that the *plays* relation connecting objects with roles is an m:n relation, which should be usable dynamically at run-time, cf. [6].

Unlike many other approaches, we argue that an explicit dependence of roles on *relations* (instead of contexts, systems, etc.) should be avoided if relations themselves are available as specific modeling elements, because of an imminent confusion with relational roles (for which a new name may be needed).

[3] For *pure roles*, see [1, sect. 3.10].

References

1. Loebe, F.: Abstract vs. social roles – towards a general theoretical account of roles. Applied Ontology (in press 2007)
2. Loebe, F.: Abstract vs. social roles: A refined top-level ontological analysis. In Boella, G., Odell, J., van der Torre, L., Verhagen, H., eds.: Proceedings of the 2005 AAAI Fall Symposium 'Roles, an Interdisciplinary Perspective: Ontologies, Languages, and Multiagent Systems', Arlington, Virginia, Nov 3-6. Number FS-05-08 in Fall Symposium Series Technical Reports, Menlo Park (California), AAAI Press (2005) 93–100
3. Loebe, F.: An analysis of roles: Towards ontology-based modelling. Onto-Med Report 6, Research Group Ontologies in Medicine, Institute for Informatics, University of Leipzig (2003) Master's Thesis.
4. Herre, H., Heller, B., Burek, P., Hoehndorf, R., Loebe, F., Michalek, H.: General Formal Ontology (GFO) – A foundational ontology integrating objects and processes [Version 1.0]. Onto-Med Report 8, Research Group Ontologies in Medicine, Institute of Medical Informatics, Statistics and Epidemiology, University of Leipzig (2006)
5. Masolo, C., Vieu, L., Bottazzi, E., Catenacci, C., Ferrario, R., Gangemi, A., Guarino, N.: Social roles and their descriptions. In Dubois, D., Welty, C., Williams, M.A., eds.: Principles of Knowledge Representation and Reasoning: Proceedings of the 9th International Conference (KR2004), Whistler, Canada, Jun 2-5, Menlo Park, AAAI Press (2004) 267–277
6. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data & Knowledge Engineering **35**(1) (2000) 83–106
7. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. 2 edn. Addison Wesley, Reading, Massachusetts (2005)

Structure and Function - Roles as the Connecting Concept

Holger Mügge

Institute of Computer Science III, University of Bonn
Römerstr. 164, 53117 Bonn, Germany
muegge@cs.uni-bonn.de

Abstract. As Riedl states in [1], p. 10, structure and function seem to be tied together forming a dichotomy[1]. Function requires structure, and structure always has a function in the sense that it affects some thing. We can distinguish two main ways of dealing with both: analyzing structures to understand functionality and synthesizing them to construct functionality. Roles play a central role both, as part of structure and as aggregate of function.

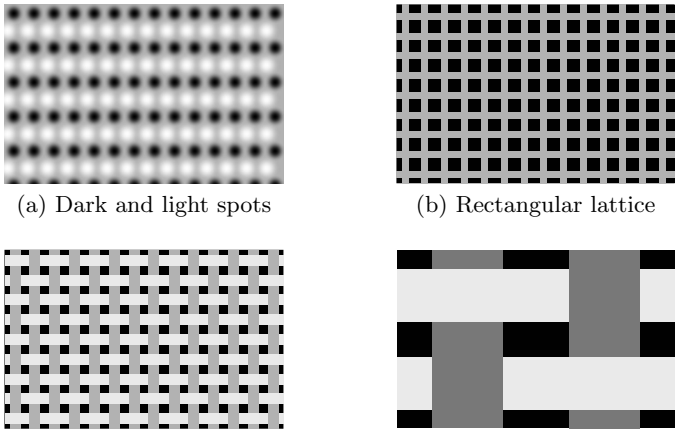
Analysis

Structure is what we as humans perceive first and easily. Function is what we have to observe consciously. One might say, we perceive structures and explain functionalities. I try to demonstrate this with a visual example, a piece of a carpet made of coarse jute threads as shown in figure 1.

Fig. 1. A piece of fabric made of jute

Now let's observe ourselves in slow-motion while we perceive the material and start to think about it. One of our first perceptions is probably the regular arrangement of lighter and darker spots (cf. fig. 2 (a)). The next step might be to detect a lattice made of strings crossing rectangularly (cf. fig. 2 (b)). A short moment later, we might recognize that the thread overlappings alternate in a further regular manner (cf. fig. 2 (c)) and concentrate on this. The latter is shown in fig. 2 (d) at a larger scale.

[1] With the exception of physics on a microscopic level where waves and particles conceptually glue together.



(a) Dark and light spots (b) Rectangular lattice

(c) Woven strings (d) Alternating overlappings

Fig. 2. Perception of the fabric structure

Our perceptions give rise to role candidates while the structure evolves. First, the light and dark spots evoke a role, let's call it "connections". Then, the straight lines of strings we perceive give us a second role candidate ("lines"). Both "connections" and "lines" become role candidates because they occur not only as unique elements but as repetitive figures and in a regular manner.

Up to now, we consider structure only. Let's assume, we start to think about why the jute threads do overlap and in such a regular manner. Now, a more conscious and analytical process starts leading us eventually to some not so obvious insights about textile manufacturing. Basically, the overlappings ensure the cohesion of the fabric. That's where function comes into play.

Summing up, beside the fabric as a whole, we identified strings playing the role of basic material, and overlappings[2] playing the role of keeping it together. I want to point out here in (our) analysis *roles show up naturally and serve as anchors of functionality.*

But roles are not the only "source"[3] of function, often only a specific cooperation of roles causes the functionality. The function of cohesion is not supplied by single overlappings. If all vertical strings were lying on top of all horizontal strings, they would not form a fabric. It is the regular distribution, the alternation of ups and downs of the string crossings in relation to each other that provides a cohesive fabric. Hence, we must consider relation also as "source" of function.

[2] One could argue that the overlappings should be modeled as relations between strings rather than as roles in their own right. But how then could one describe their regular distribution on a grid?

[3] I hesitate to describe roles as "source" of function. Other words equally mistakable are: "location", "home", or "origin".

Synthesis

Now, we take the opposite direction. Assume, our task is to build a mat out of some given thin ropes. Without further knowledge this might easily be frustrating. We depend on the idea of weaving[4]. There is no incremental and contiguous way to get to the solution and build a coherent fabric out of strings. We would probably prefer to agglutinate some leafs together or go hunting bears to get their skin, if we do know the structure of woven fabric and in particular the role strings and their overlappings can play.

Of course, many tasks do not require such inventional ideas and can be achieved instead by combining more obvious solutions. But in principle the situation is often the same, we need knowledge in form of structures made up of roles and their interrelations.

Knowledge

The little example presented in the previous sections illustrates two "facts" about knowledge: first, gathering already applied knowledge by analysis can be driven by structural decomposition. The fabric analysis is relatively easy because its structure can be disclosed visually, a task that inheres human cognitive capabilities. Roles and the relations between them can guide the the observer to understand the system he investigates.

Second, finding not so obvious solutions often has a footprint in the structure of the constructed object or system. It is usually difficult to reconstruct this idea from the solution as in our example. This strongly asks for support knowledge transfer and persistence by making structures visible, in particular if their not visual by nature. And it also holds when the idea is not newly invented but reused.

Roles can provide the vocabulary of the idea behind the solution on an abstract level, that allows for mapping the solution to largely different situations. E.g. in the fabric example, the role of the threads might be taken by metal rods , when instead of a mat the system is an iron gating covering a basement window. Further abstraction of this structure might leave solely a grid. Hence, abstraction needs to be applied with care, when the resulting structure shall remain meaningful. Piaget gives in [3] an example of successful abstraction that leads to a widely applicable structure which still remains significant, namely the mathematical concept of a group. He calls the process that has led to it *reflective abstraction.*

[4] "Archeologists believe that basket making and weaving were probably the first "crafts" developed by humans. [...] there is evidence of cloth being made in Mesopotamia and in Turkey as far back as 7000 to 8000 BC." (cited from [2])

Software-Engineering

In most natural sciences analysis is more prominent than synthesis. Not so in computer science. Here, analysis relates to readability of programs, understandability of designs etc. But these issues apply only afterwards, i.e. when the program already has been constructed. Hence, synthesis is the more valued[5] issue.

Software typically is "written" as a large collection of texts. Its structure is not easy to visualize. Therefore, supporting the perception of structure is an important task. It must be performed during construction to save the knowledge of the innovations.

Prominent examples of such structural knowledge in software-engineering are software patterns (cf. [4]). They typically describe structures, give them a name, describe the most important roles, and provide examples of their usage. Relations are often less clearly specified. Beside making the discovery or invention behind the pattern reusable it allows us to speak in terms of the language it provides by naming the roles it comprises (cf. [5]).

Some further Statements

- Roles without relations do not form a structure. Single roles are nothing but abstraction of function.
- Roles are tightly interconnected with symmetry. They provide the variable parts of structure without changing the structure.
- The structural rules can rely upon relations or roles. In the latter case the feature of roles as connection between structure and function becomes particularly obvious[6].

References

1. Riedl, R.: Strukturen der Komplexität. Springer, Berlin (2000)
2. Wylly, S.C.: The Art and History of Weaving. Georgia College & State University, (http://www.faculty.de.gcsu.edu/ dvess/ids/fap/weav.html)
3. Piaget, J.: Le structuralisme. Press Universitaire de France (1968)
4. Gabriel, R.P.: Patterns of Software. Oxford University Press (1996)
5. Unger, B., Tichy, W.F.: Do design patterns improve communication? In: Workshop on Empirical Studies of Software Maintenance (WESS), http://hometown.aol.com/geshome/wess2000/metricsandmodels.htm (2000)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1994)

[5] In my opinion, not only the relative valuing of analysis and synthesis is in computer science inverse to most natural sciences, but also the tendency. While e.g. biology in term of genetics gets more and more a synthesizing flavor, computer science nowadays constructs systems that are so large, that analysis becomes more and more important.

[6] A concrete example of the area of software-engineering is the notifier role in the observer pattern (cf. [6]), which is typically realized by a type's method.

Roles and Classes
in Object Oriented Programming

Trygve Reenskaug

University of Oslo
Department of informatics
PO box 1080 Blindern, 0316 Oslo, Norway.
http://folk.uio.no/trygver
trygver@ifi.uio.no

Abstract. The value of a system is greater than the sum of its parts; the system organization giving the added value. In this article we show how a system description can be split into state and behaviour parts. State is described in terms of classes and associations; behaviour is described in terms of roles and connectors. This article focuses on the behaviour part and its most important contribution is the use of selection to bind roles to objects and thus to classes. This leads to a balanced paradigm for describing the state and behaviour of object systems.

Key words: object systems, collaboration, role, object, class, data model.

1 Introduction

Objects encapsulate state and behaviour. Object state is represented in the object's instance variables. Object behaviour is specified in the object's methods. The execution of a method is triggered by the object receiving a message. The binding of message to method is dynamic and depends on the implementation of the receiving object.[1]

The value of a system is greater than the sum of its parts. The parts have their own intrinsic value, and the added value stems from the system organization. The properties of a system are similar to the properties of an unattached object. System state is the accumulated state of its objects and their associations. System behaviour is triggered by messages that the system receives from its environment and is accomplished through an organized process of message interaction between its objects.

Current mainstream programming languages are class oriented; they specify sets of objects with common properties. Class based languages work well in simple cases; but they are less than ideal in complex cases where the system as a whole tends to be hidden among the details of the classes and methods. In this article, we remedy this deficiency by showing how class centred programming can be augmented by role centred programming where system behaviour is specified explicitly in terms of collaborations and roles.

1. In some cases it also depends on the nature of the sending object.

The role is a slippery concept. Roles cannot be defined by their shape or their constitution, only by what they do in the context of a system. *The essence of object orientation is that a system of objects collaborate to achieve a common goal.* Roles are references to participating objects; each role represents the contribution these objects make towards a system goal

The work reported in this article is part of a project we call *BabyUML* . The project goal is to create a programming environment with explicit specification of system state and behaviour; this article describes the conceptual underpinnings of this environment. The project motivation and initial ideas are presented in [1]. A project status report will appear in [2].

Section 2 introduces the *collaboration* as a context for roles. Section 3 builds an intuition about the nature of roles based on analogies with common usages of the word. Section 4 defines the concepts of *role* and *class* as they are used in BabyUML. Finally, we conclude in section 5 by summing up how roles and classes give two independent perspectives on systems of interacting objects.

2 The Collaboration

A system of interacting objects is called a *collaboration*. Its primary purpose is to describe how the system works when performing a task. Only those aspects of the objects that are required for the task need to be described. Thus, details, such as the identity or precise class of the actual participating objects are suppressed.

The UML collaborations stem from the *OOram role modelling* technology. We first used it in our own development work from the early eighties; our tool was demonstrated in the Tektronix booth at the first OOPSLA in 1986. It was first mentioned in print in an article by Rebecca Wirfs-Brock [3], and later in a JOOP article [4]. A full treatment including the parts that have still not made it into UML is in [5]. Egil Andersen gives a theory of role modelling with special emphasis on system behaviour and model inheritance (synthesis) in [6]. We use the term *collaboration* in the BabyUML project rather than *role model* because we are into programming with roles, not merely modelling with them.

As a collaboration example, consider the *Observer pattern* from the *Design Patterns* book [7]. This pattern maintains consistency between related objects called `subject` and `observers`. The `subject` is an object that holds some state. The `observers` are objects that need to be informed about any change to this state.

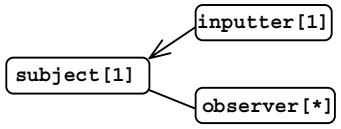


Figure 1 The Observer collaboration.

Figure 1 show the Observer pattern as a collaboration. There are three roles that represent the participating objects. The role names are aliases for the objects that will occupy the respective positions in the structure during an execution.

Figure 2 shows how the Observer collaboration maintain consistency between the `subject` and `observer` objects when the state of the `subject` object is changed by a stimulus from an `inputter` object. The diagram is a BabyUML sequence diagram that describes sequential interaction. A filled arrow denotes message transmission. A thin, vertical rectangle denotes a method execution. Any number of objects may be bound to the `observer` role; they work in lock-step so that their updates appear to occur simultaneously.

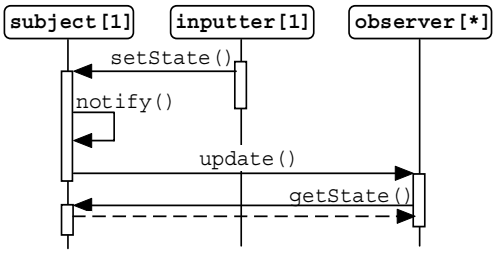


Figure 2 The Observer pattern interaction

Many objects may play the `observer` role in different contexts and at different times, but we are only concerned with the objects that play the role in an occurrence of the interaction. A role may be played by many objects and an object may play many roles. In this example, an object playing the `inputter` role often also plays the `observer` role. The object diagram in Design Patterns [7] mandated this by showing two objects called `aConcreteObserver` and `anotherConcreteObserver` respectively; the first also playing the `inputter` role.

Every message has a sender and a receiver, both are equally important from the perspective of system design. The collaboration reflects this; there are no messages without both a sender and a receiver. This in contrast with classes where the handling of received messages is specified explicitly, while the sending of messages is hidden within the methods.

The BabyUML project has UML in its name to indicate that we see UML [8] as a rich source of alternative and consistent ways of describing system behaviour; the sequence diagram of Figure 2 is a specialization of one of them.

3 The Word *Role* in Common Usage

We will now look at how the word *role* is generally used and see how this usage can help us build an intuition about our use of the word in object oriented programming.

Most people will probably associate the word *role* with a part played by an actor in a theatrical performance. Oxford [9] gives its origin: *"from obsolete French roule 'roll', referring originally to the roll of paper on which an actor's part was written."* (Note that this means the script, not the performance!) Oxford [10] has a fitting definition: *"Actor's part; one's function, what one is appointed or expected or has undertaken to do."* This maps nicely on to our use of roles for naming the participants in a system of collaborating objects; the role is a reference to an object[1]; it represents the object's function, what it is delegated or expected or is required to do.

A collaboration is analogues to a play where its code is analogues to the written drama. A role is a part in a collaboration just as a role is a part in a play. An Object plays a role in an execution of a collaboration; an actor plays a role in a particular performance of a play. An Object is selected to play a particular role at a certain time and in a certain context. An actor is selected to play a particular role in a particular performance.

An actor interprets a given role in his personal way; other actors do it differently. An object playing a particular role is selected from a pool of candidate objects. These objects may play the role differently because they can be instances of different classes even if they all satisfy the role's requirements.

Steven Pinker in *How the Mind Works* [11] claims that the meaning of a system comes from the meaning of the parts *and* from the way they are combined. Consider this very simple example taken from the Pinker's book:

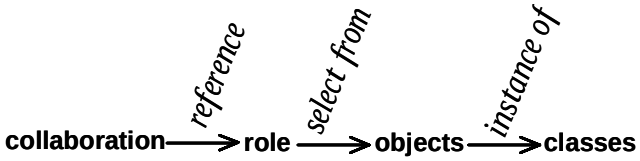
> The baby ate the slug
> The slug ate the baby.

Each of the three words "baby", "ate", and "slug" has its own meaning. The two diametrically different meanings arise from the roles played by the words in the sentences. Analogically, an object system gets part of its meaning from its objects. It gets the rest of the meaning from the roles played by the objects and the way those roles are organized. As an example, consider how the Observer pattern is described in terms of roles in Figure 1 and Figure 2. The roles tell a part of the story; it is only completed when the roles are bound to actual objects.

4 Roles and classes -- a Set of Definitions

The value of a system of interacting objects stems partly from the value of its objects taken separately, and partly from the way the objects are organized to represent system state and behaviour. Figure 3 illustrates how a system can be organized to realize its behaviour. A collaboration describes how the system performs a task. Objects are represented by the roles they play as they make their contribution towards this task. These objects are selected at run time from a pool of objects by some selection mechanism. The selected objects are instances of one or more classes, thus completing the bridge from the performance of a task via roles to classes.

Figure 3 The bridge from role to class



The binding between role and object is dynamic because an object may play different roles and a role may be bound to different objects. The result of the selection can depend on many things such as the nature of the task, the process itself, and the current state of

1. One role can reference several objects simultaneously. They will all serve the same purpose in the Collaboration, but we can assume a single object without loss of generality.

the objects. Contrast with the static binding between object and class; an object is an instance of the same class throughout its lifetime. The notions of role and class are orthogonal. A role represents the usage of an object while the class represents its construction.

We will now propose a definition for each of the parts in Figure 3. We end in section 4.5 with a discussion of the dynamic step of binding roles to objects.

4.1 The Object

The *object* is the atom of object oriented data processing systems. Objects encapsulate state and behaviour and have the following properties:

- *Identity:* An immutable property that distinguishes an object from all other objects.
- *State:* The object's attributes and their values.
- *Interface:* The set of messages that can be received by the object.
- *Behaviour:* The methods that specify the object's processing of incoming messages. This includes changing the object state and sending messages to itself or other objects.
- An object is an instance of the same class throughout its lifetime.

4.2 The Role

The concept of *role* conforms to the common usage of the word:

- A role represents a functionality.
- This functionality can be utilized in its collaboration with other roles.
- A role is bound to one or more objects selected from a universe of objects. These selected objects are called the *players* of the role.
- A role delegates the performance of its functionality to its players.
- A role specifies requirements for its players. An important property is the set of messages that its players must understand.
- The required properties can be implemented by several classes so the players need not be instances of the same class. Different objects can thus perform the same role in different ways.

4.3 The Collaboration

A *collaboration* is a structure of interconnected roles that enables the system to perform one or more tasks.

- A collaboration describes a structure of collaborating roles, each performing a specialized function, which collectively accomplish some desired functionality.
- A collaboration structure constitutes a graph where the nodes are roles and the edges are the message interaction paths.

4.4 The Class

A class is a specification of the state and behaviour of objects. In the context of a system, an object's state is part of the system state and an object's behaviour is part of the system behaviour.

- The state is specified as a set of attributes.
- The behaviour is specified as a dictionary binding messages to methods where a method is a piece of executable code.
- The instances of a class are objects. They are all different, but they share the features described by the class.
- Classes can be organized in an inheritance hierarchy. This hierarchy is independent of the message interaction structures defined by the collaborations.

4.5 Binding roles to objects by dynamic selection

The concepts discussed above are all well known and there are well established standards for applying them to software design. UML 2.x [8] can be taken as a starting point. Roles are here defined as properties of collaborations in UML; this corresponds well with our notion of a role. UML class diagrams can be used to model the system data.

The crucial point in Figure 3 is the link between *role* and *objects*. It is annotated by *select from*; this signifies that objects are dynamically selected from a set of relevant objects to play the roles. Many different selection mechanisms can be used. We will here consider two; the first is a very simple one and the second more general.

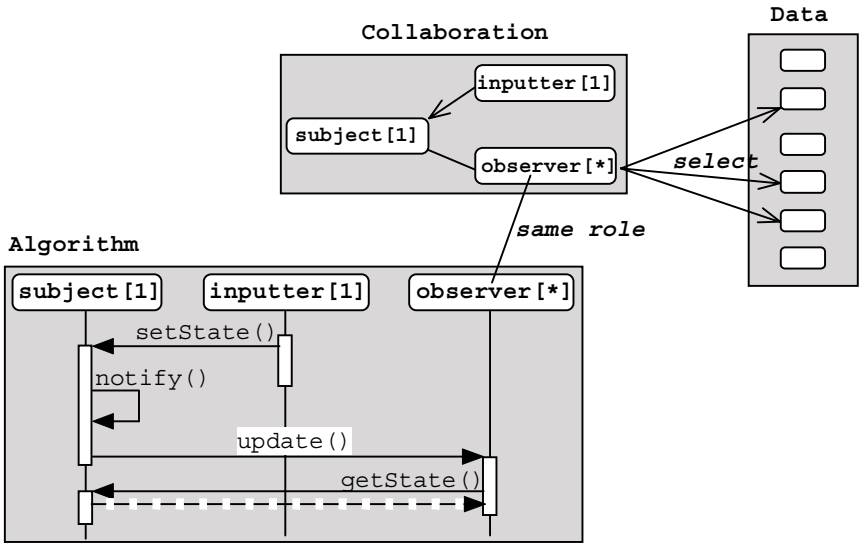
Consider the Observer pattern discussed in section 2. The pattern in [7] specifies that the `subject` has a list of `observers`, each conforming to the simple interface needed to play that role. This means that an object playing the `subject` role must understand messages such as `addObserver()` and `removeObserver()`. The selection mechanism is very simple; any object that happens to be in the list of `observers` will receive the `update()`-message.

A more elaborate example is based on the *Data-Collaboration-Algorithm (DCA)* paradigm, a paradigm that makes the collaboration a first class program element [2]. Figure 4 illustrates it with a variant of the Observer pattern:

- The *Data* part is responsible for knowing all relevant objects. In a typical application of the Observer pattern, they can be model, view, and controller objects in the MVC paradigm. (In Java, the Data part can be implemented in a class defining a "micro database", see [2].)
- The *Collaboration* part is an object that has a method for each role. These methods dynamically select the appropriate player objects. In principle, the methods should perform the selection on each call to ensure up-to-date mapping. (In [2], the nature of the example is such that the result of the query depend on the state of the objects and this state changes on every execution of the Algorithm).
- The *Algorithm* part is responsible for managing the flow of messages during an interaction. The algorithm references roles by name, and delegates to the collaboration to bind the roles to objects. In our variant of the Observer pattern, the code could look like this:

```
for (Observer obs : collab.observers()) {
    obs.update();
}
```

Figure 4 The DCA paradigm (Data-Collaboration-Algorithm)



To make the example more concrete, imagine a *Dog Breeders Association* that keeps a registry of its of pedigree dogs. There are also dog shows where dogs receive prizes according to their merits. After a show, the registry dog records shall be updated with prizes received. We let the `subject` role be played by an object that represents the show and let the `observer` role be played by the registry objects that represent the prized dogs. The code would first set

```
subject := show
```

and then select the `observers` from the dog registry (here coded in an unspecified pseudo-language):

```
define observers as
    select prized from dogregistry
    where priced is in subject.winnersList
```

The algorithm can then simply augment each `observer` with the relevant prizes from the `winnersList`.

One advantage of the Observer pattern defined in [7] is that it provides loose coupling between `subject` and `observer`. This DCA variant is even looser; the `subject` does not know the `observers` and the `observers` do not know the `subject`. The link between them only exists during the transfer.

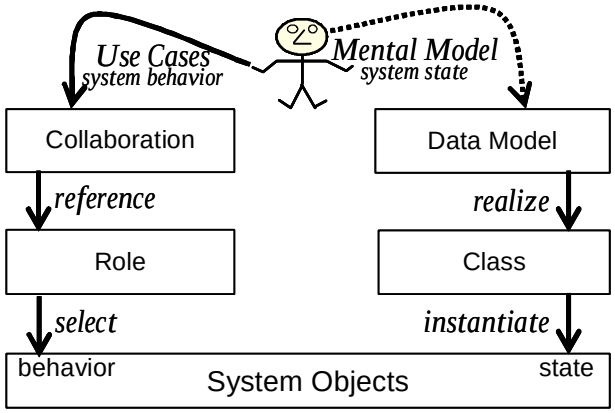
5 Conclusion

Objects have state and behaviour; they are specified separately in the instance variables and in the methods. Systems also have state and behaviour; we separate their specification in the two paths of shown in Figure 5. *System state* is specified in a *Data Model*; e.g., in the form of a UML class diagram showing the classes with their attributes and associations. The classes specify the attributes (state) of the objects; the associations are specified in the Data Model. *System behaviour* is specified in the system's collaborations. The collaboration roles specify the requirements that any object playing that role must satisfy. Finally, the methods in the participating objects are implemented in the corresponding classes so as to satisfy these requirements.

We see that the attributes of a class stem from the system's Data Model; while the methods of a class stem from the behaviour required for its instances.

The system environment is here symbolized by a human user. The way we describe systems is recursive; the environment could have been external objects and our system would then have been a subsystem.

Figure 5 Roles and classes in object oriented programming



The architecture of Figure 5 can be implemented in a language such as Java. The problem is that there is no Java language construct for explicitly specifying roles and collaborations. The consequence is that their implementation will be scattered around in the code.

The goal of the BabyUML project is to make system state and behaviour explicit by extending object oriented programming with additional projections. *Programming with Roles and Classes; the BabyUML Approach* [2] reports on the current state of the project. It includes the results of a Java experiment and the beginnings of an IDE for the specification of systems of interacting objects.

Systems are characterized by their state and behaviour. In this article, we have focused on system behaviour and have shown how it can be described in terms of collaborations and roles. We have also shown how roles can be bound dynamically to objects and this classes. The BabyUML project wants to make system behaviour explicit in the code, but our current results are fully applicable to system modelling and design.

Steven Pinker in *How the Mind Works* [11] claims that we have two independent ways of dealing with complexity; one is grouping things according to their properties, another is grouping things according to what they are used for. We use the class for the first grouping and the role for the second. Both classes and roles are essential abstractions that we need in order to master the complexity of the world round us. In this article, we have shown that the role is the atom of system behaviour. *The role should, therefore, be granted the status of an ontological primitive.*

References

[1] Reenskaug, T,: *The BabyUML discipline of programming (where A Program = Data + Communication + Algorithms).* SoSym **5**,1 (April 2006). DOI: 10.1007/s10270-006-0008-x. [web page] http://heim.ifi.uio.no/~trygver/2006/SoSyM/trygveDiscipline.pdf

[2] Reenskaug, T.: *Programming with Roles and Classes; the BabyUML Approach;* A chapter in *Computer Software Engineering Research;* ISBN: 1-60021-774-5; Nova Publishers; Hauppauge NY; 3rd Q. 2007; [WEB PAGE] http://folk.uio.no/trygver/2007/babyUML.pdf

[3] Rebecca J. Wirfs-Brock , Ralph E. Johnson: *Surveying Current Research in Object-Oriented Design.* Comm ACM **33**(9):104-124. September 1990.

[4] Reenskaug, T.; et.al. *ORASS: seamless support for the creation and maintenance of object-oriented systems;* JOOP, **5** 6 (October 1992); pp 27-41.

[5] Reenskaug et.al.: *Working with objects. The OOram Software Engineering Method;* ISBN 1-884777-10-4; Manning, Greenwich, CT. 1996; Out of print. Early version in [web page] http://heim.ifi.uio.no/~trygver/1996/book/WorkingWithObjects.pdf

[6] Andersen, E. P.; *Conceptual Modeling of Objects. A Role Modeling Approach.* D.Scient thesis, November 1997, University of Oslo. [web page] http://heim.ifi.uio.no/~trygver/1997/EgilAndersen/ConceptualModelingOO.pdf

[7] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J,; (GOF) *Design Patterns*; ISBN 0-201-63361-; Addison-Wesley, Reading, MA. 1995.

[8] Unified Modeling Language: Superstructure. Version 2.1.1; Object Management Group; document formal/2007-02-05; [web page] http://www.omg.org/cgi-bin/doc?formal/07-02-05.pdf

[9] *AskOxford.* Oxford Dictionaries.[web page] http://www.askoxford.com/?view=uk

[10] Fowler, H. W.; Fowler, F. G.; McIntosh, E.: *The Concise Oxford Dictionary of Current English.* Fifth Edition; ISBN 0 19 861107 2; Claredon Press, Oxford 1975.

[11] Pinker, S.; *How the Mind Works*; ISBN 0-393-04535-8; Norton; New York, NY, 1997.

Acknowledgements

The author is very grateful to James O. Coplien for his continued interest over many years, his valuable insights, and not least of all, our very challenging and productive discussions. The author is also grateful to an anonymous reviewer for valuable comments.