A PRECISE SPECIFICATION FOR THE MODELING OF COLLABORATIONS

Fernando Valles Barajas

Department of Information Technology, Faculty of Engineering, Universidad Regiomontana,
15 de Mayo 567 pte., C.P. 64000 colonia centro, Monterrey, Nuevo León, México,
Tel. +52 81 82204733,
email: fernando.valles@acm.org, fernando.valles@ieee.org

ABSTRACT

A collaboration represents a set of entities that work together to achieve a common goal. Collaborations are useful to specify how a set of elements collaborate to realize a method, a class or a use case. Collaborations are also useful for representing design patterns, which are represented in UML as parameterized collaborations. This paper contains two complementary models for modeling collaborations; one model is graphical and specifies only basic constraints and the other model is textual and specifies further constraints that cannot be specified in the graphical model. Both of these models were built using Alloy, which is: 1) a modeling language that uses first order logic and relational logic to specify systems 2) a methodology that helps designers in making models 3) an analyzer that detects inconsistencies in design.

Keywords: Formal methods, Alloy, UML, Collaborations.

1.0 INTRODUCTION

UML is a graphical modeling language that uses several diagrams to get different views of a system [20]. Two UML diagrams that complement each other are class diagrams and sequence diagrams. Class diagrams allow one to specify a set of entities and their relations. Sequence diagrams specify how the entities, specified in a class diagram, interact with each other by passing messages to achieve a specific goal.

A collaboration can specify how entities that work together to achieve a specific goal are related as well as how they interact. Collaborations are represented in UML using structured classifiers, which are classifiers with an internal structure. In the internal structure of a structured classifier the entities that collaborate and their relations are drawn. Collaboration can be used:

- To specify and document how a set of entities realize a use case, a classifier or an operation.

- To document design patterns, which are common solutions to frequent problems found in software systems [4]. A design pattern is represented in UML using parameterized collaborations.

Motivation and contribution of the paper:
Formal methods are used to make software models without ambiguity using first order logic and relational logic [8]. Lightweight formal methods, in particular, are very attractive tools because they build analyzable models [11].

In this paper two models were built to model collaborations; these models formally specify both how the entities that play roles in a collaboration are related and how these entities interact with each other. These models can be used as a guide in building a collaboration modeling tool; if a person wanted to build such a tool, she/he would know the exact rules that govern collaborations.

The models are made using a lightweight formal method called Alloy, which is also a modeling language and a model analyzer.

Related works:
In [16] the author presents a precise definition of the realization of use cases made by collaborations. The author also formalizes the concept of sound and complete subsystems. The author argues that his work will serve as a first step for building a tool that supports checking of the soundness and completeness of a system. The author remarks that it is necessary to formally define collaborations to avoid ambiguities and misunderstandings in the

definitions related to this concept. The paper focuses on the formal definition of use case realizations; it does not consider the formal definition of classifier and operation realizations or the precise definition of design pattern as a special kind of collaboration. Using the UML notation, some graphical models of the collaborations are presented. The following concepts were not modeled: collaboration instances, collaboration use and interaction instances. Only one relationship between collaborations was considered: generalization.

The paper [16] and this paper have the same motivation: to serve as a support in building a collaboration modeling tool. That paper only models the realization of use cases by collaborations; the realization of operations and classes are not included. Modeling design patterns as structured collaborations is not included either. That paper uses the UML notation to present some graphical models of collaborations while this paper uses the Alloy notation, which is simpler and easier to understand.

In [7] the authors present a formal definition for a collaboration refinement technique. As described by the author, "collaborations are used to describe the realization of services represented by use cases or operations". A service is represented by a set of interactions. A collaboration refinement should provide a more detailed description of the interactions in the context of the collaboration structure. The process of refinement can be used to link the elements of one model on a more abstract level, to the elements of another model on a more specific level; for example, these models can be a model obtained in the analysis stage and a model built in the design stage.

This process of refinement is useful for documenting changes that are made in a particular system. The motivation of the authors is that there are several ambiguous definitions (that can be found in the literature) of collaboration refinement. The authors also claim that many of the modern development methodologies of software are based on model refinement (one of these is RUP) but there is no procedure to refine collaborations and to check if one model is a refinement of another.

In that paper, collaborations are modeled as special kinds of classifiers; more specifically collaborations are modeled as structured classifiers. The concept of collaboration occurrence (the term collaboration occurrence was used in OMG UML 1.5 to refer to an instance of a collaboration, in OMG UML 2.0 this term was renamed to collaboration use), which is considered to be an instance of collaboration, is taken into account in that paper. Set theory is used for the formal description of collaboration refinement.

In that paper the authors focus in the formalization of the process of refinement; this process is used to refine a use case by using collaborations. Although this is not the main objective of this paper, the refinement of use cases, operation and classes is presented.

In [5] an analysis of the representation of design patterns as parameterized collaborations is given. The authors state that the approach used by the UML community to represent design patterns as a kind of template classifiers can lead to an ambiguous representation. The author formally specifies the concept of design patterns.

This paper includes not only a formal model of design patterns but also a formal specification of the realization of classes, operations and uses cases by using collaborations.

In [1] a method to formally specify the behavior of collaborations is presented. The specification is made using Object-Z, a modeling language based on the Z modeling language. In the method proposed by the authors, a use case model of the requirements is first made, collaborations are then used to realize use cases and finally, in the last step the collaborations are specified using Object-Z.

In [14] the 1.5 version of UML, which is a document made by the organization in charge of the UML standard, a formal and a graphical model of collaborations is presented. The graphical model is made using the UML notation and it is complemented with a formal model made with the use of the Object Constraint Language (OCL).

In that document, the multiplicity attribute of a role represents the number of instances played by a role in a collaboration. In this paper the multiplicity attribute represents the number of instances played by a role in all collaborations. In the UML 1.5 the *ownedElement* attribute represents the set of roles defined in a collaboration. These are classifier roles and association roles. For the purpose of simplicity, this paper only considers one kind of role: classifier roles and herein they are only referred to as roles.

In that paper, classifier roles and association roles are the parameters of parameterized collaborations. In this paper, roles are the only parameters for a collaboration. The following characteristic of the model is considered in UML 1.5 but it is not considered in this paper: an association may be traversed in some, but perhaps not all, of the allowed directions in the specific collaboration; that is, the value of the *isNavigable* property of an association end may be false even if the value of that property of the base association end is true.

In [15] the 2.1 version of UML is presented. That document does not contain a graphical model for collaborations and the formal model for the collaboration is smaller than the formal model presented in version 1.5 of UML. Upon comparing [15] and [14], the following can be concluded: the most important change is that the concept of collaboration is introduced as a composite structure. From the UML 1.5 to the UML 2.1, the Unified Modeling Language has suffered some changes; for example, the term *association*, which is a link between roles, was renamed *connector*. The concept *collaboration instance* was renamed *collaboration use*. The notation for drawing collaborations and collaboration uses is introduced in the UML 2.1. The concept of role realization through interfaces is added in the UML 2.1.

In [19] a new paradigm of programming, called BabyUML, based on the concept of collaborations is introduced. In this paradigm a program is composed of: data structures, algorithms and communication. The author argues that the object-oriented paradigm is not enough to obtain a readable and clear code when building complex systems. Objects that have some logical relationship are specified in a component, which is a concept introduced in BabyUML to deal with the complexity of software. BabyUML allows one to specify the way objects interact with each other by using the concept of collaborations. This paper remarks the importance of the collaboration concept in building software.

Structure of the paper:
In this section the motivation and contribution of the paper and related works have been given. The following section contains some basic concepts of the formal method that will be used to model collaborations: Alloy. Section 3 introduces the necessary concepts of collaborations for understanding the models proposed in the paper. Section 4 presents a metamodel, which is used as an overview of the formal model of collaborations. Section 5 contains the formal model. Concluding remarks are given in section 6.

2.0 ALLOY: A LIGHTWEIGHT FORMAL METHOD

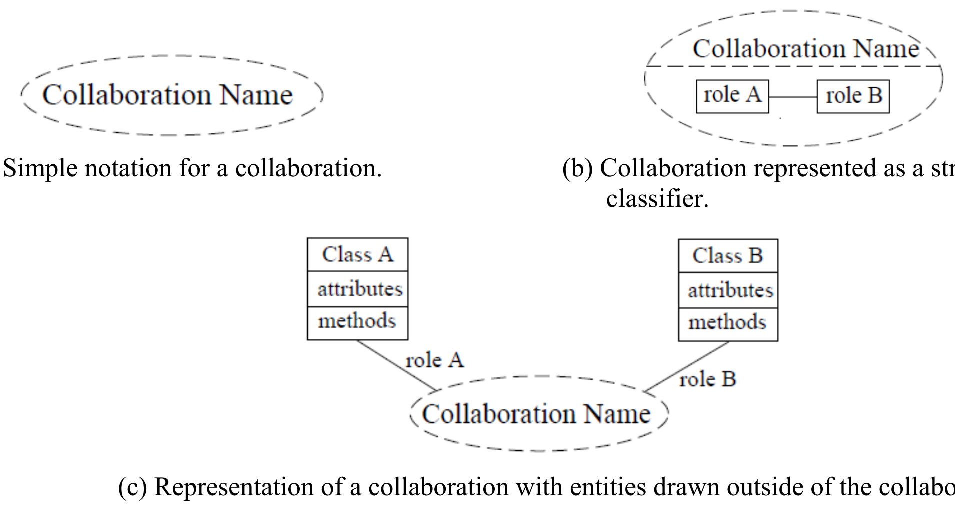
Formal methods are tools that use mathematical notations to make software models [21]. Alloy is a formal method but it is also a modeling language and a model analyzer [10]. This modeling language uses first order logic and relational logic to make software models. The advantage of using first order logic instead of higher order logic is that it is possible to perform an automatic analysis of software models [9]. Another advantage of Alloy is that models written in this modeling language are made by using ASCII characters; so it is not necessary to have a special tool to type an Alloy model.
An Alloy model is composed of two sub-models: a graphical model, which represents an overview of the system, and a textual model, which specifies further constraints on the graphical model. To make a software model, Alloy specifies the entities of a system by using signatures, which are similar to classes in the object oriented theory, and defines the relations between these entities as attributes of the signatures. Multiplicity keywords can be used in a textual model to limit the number of elements that a signature can have and they can also be used to constrain the number of elements of a signature that participates in a relation. Multiplicity keywords are: **one**, **some, lone**, and **set** which respectively mean only one (the default), one or more, zero or one and any number. In a graphical model the designer can also define similar constraints using multiplicity symbols.

The multiplicity symbols are: !, +, ?, * and their meanings are: one, one or more, zero or one and any number (the default).

3.0 COLLABORATIONS

A collaboration represents a set of entities that work together to provide a desired functionality that is greater than the sum of the functionality of all entities [12]. This concept was born in UML 1.0 [3]. While reading the definition of collaboration, the reader may think in a sequence diagram, but as it will be seen later, collaboration entities are modeled by the roles they play and a collaboration is also made at a higher level of abstraction.

There is no UML diagram to model collaborations, however UML has a special notation to model collaborations which is discussed as a part of composite structures [3]. A collaboration is represented using a dashed ellipse with the name of the collaboration written inside, see fig. 1(a). The name of the collaboration should be a simple noun phrase that communicates the essence of what the collaboration does [12] and it can include the package that contains the collaboration [2]. The first letter of a collaboration name is usually capitalized (for example the collaboration name of fig. 1(a)).



(a) Simple notation for a collaboration.

(b) Collaboration represented as a structured classifier.

(c) Representation of a collaboration with entities drawn outside of the collaboration.

Fig. 1: Styles to represent a collaboration.

There are two styles for representing the participants involved in collaborations:

1. The first is based on drawing a compartment inside the collaboration and then the participants of the collaboration and their associations are drawn, see fig. 1(b). The associations are drawn using solid lines, which are called connectors, and the participants are labeled with the role that they play. The reader should notice that the names of participants are not capitalized; they are not classes. This style is used with the assumption that a collaboration in UML is a kind of structured classifier, which is a classifier with an internal structure; so the internal structure for a collaboration are the participants and their associations.

2. The second is based on drawing the participants outside of the collaboration ellipse. The role that each participant plays is written on a line that connects a participant with the collaboration, see fig. 1(c). Using this style the role that the participants play in the collaboration is indicated in the communications links. The advantage of this style is that the attributes and the operations of each participant can be specified; the disadvantage is that the direct associations between the participants are not specified [18]. A detailed explanation of fig. 1(b) is given as follows; in this figure a connector defines a relationship between the entities that play roles in the collaboration. Connectors are not directly mapped with the associations of a class diagram; the participants in a collaboration connected by connectors need to be related with the same composite object. In a class diagram this constraint does not apply.

Communication with the external world and the structured classifier is modeled by using ports. Every port attached to a structured classifier may have a set of interfaces each having provided or request services.

A collaboration has two parts: the static part and the dynamic part. The internal structure of a collaboration represents its static part and the dynamic part is modeled using some kind of interaction diagram (a sequence or a communication diagram) or any other behavioral diagram like a state machine [18].

A collaboration is defined in terms of the roles played by entities; if one entity wants to play a specific role, it must conform to that role, meaning that the features defined in the role are a subset of the features defined in the participant; so it is legal to specify every participant using an interface [15].
The participants in a collaboration can be defined as placeholders for objects; these objects have to conform to the roles played by the participants.

The names of the participants in the collaboration do not follow the rules for naming classes; it is necessary to label the roles played by entities. The convention for the name of a participant in a collaboration is: participant-name/role-name: class-name, where participant-name is the name of the instance that plays the role labeled as role-name and class-name is the type of participant. All of these parts are optional [17]. This notation is useful because by using a standard notation to label the participants with their roles, it not necessary to attach notes with this information to each participant [3], [6].

A connector, which denotes the cooperation between two participants, can be implemented by an association of a class diagram but this is not a general rule; the relationship between two participants can be transient. For example, participants in a collaboration that realizes a method (which are the procedure parameters, the local variables and the global variables that are accessible to the method) collaborate to implement the functionality of the method when it begins but once the method finishes these elements may not collaborate.

The instances that participate in a collaboration can exist before and after the collaboration.

Roles and connectors may have types (classifiers and associations between classifiers); the association type is optional, because as was stated previously, relationships between the entities can be transient.

An object can play different roles in different collaborations; however it is possible for the same object to play different roles in the same collaboration. One role in a specific collaboration can be played by different objects.
A collaboration can be used to realize a use case, a class or a method. The notation used by UML to represent the realization is to draw a dashed straight line with a large unfilled triangular arrowhead, pointing to the element realized by the collaboration, see fig. 2.

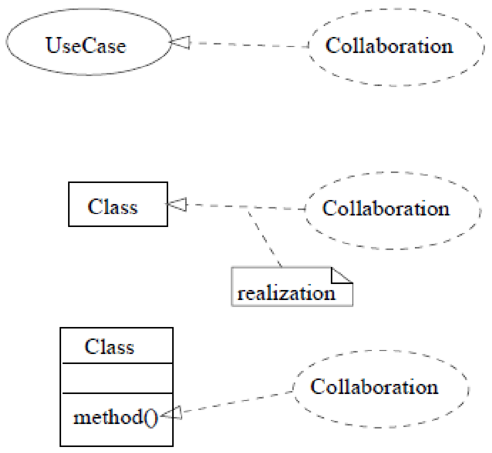


Fig. 2: UML notation for specifying subjects realized by collaborations

Collaborations can also be used in UML to represent design patterns, which are solutions to a common problem in a given context. This form for representing design patterns is seldom used for designers [3].
According to [20] a pattern is a parameterized collaboration with its related documentation; this definition was taken to model the design patterns, as can be seen in fig. 3.

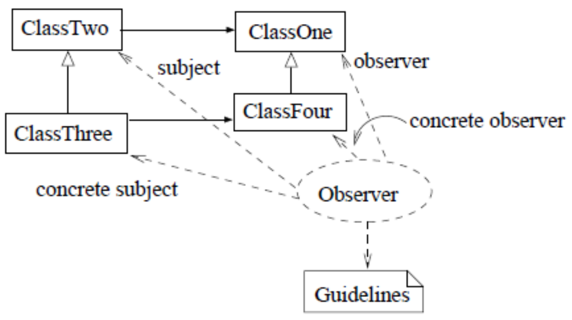


Fig. 3: A design pattern represented as a parameterized collaboration

This figure shows an example of a design pattern represented as a parameterized collaboration. There is a dotted line for each class that plays a role in the collaboration; this line is labeled with the role the participant plays. For example, the dotted line connected to the *ClassOne* class indicates that this class plays the role of observer. Each of these dotted lines is a parameter of the design pattern. Guidelines to use the pattern can also be attached to the pattern (see fig. 3). A collaboration can be used several times to realize different subjects (use cases, operations, classes); every time a collaboration is used to realize a subject, a collaboration use is defined. A collaboration use defines a link between every role defined in a collaboration and every instance defined in a subject. Each one of the instances must conform to a role defined in the collaboration; which means that instances must possess all the features defined in a role. The notation for representing a collaboration use is shown in fig. 4.

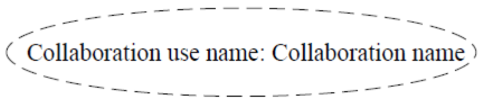


Fig. 4: UML notation for specifying a collaboration use

As it can be seen in this figure, a collaboration use is drawn using a dashed ellipse and it is labeled with the name of the collaboration use followed by a colon, and then the name of the collaboration type; this notation resembles greatly the notation used to label an object [12]. For each role defined in the collaboration use, a dashed line is drawn to the instance that conforms to the role.

This section has explained some basic concepts of collaborations that are necessary for understanding the models proposed in this paper. The following section contains a graphical model for collaborations.

4.0 THE GRAPHICAL MODEL

In figs. 5 and 6 a model of collaborations is presented using the graphical notation of Alloy. As can be seen, this notation is similar to the notation used by UML to make a class diagram. Each box in the diagram represents an Alloy signature and each arrow denotes a relation between signatures. There are two types of relations in Alloy: associative relation, represented by a line with an open arrow, and inheritance relation, represented by a line with a hollow arrow. There is an associative relation, labeled as *interactions+*, between the signatures *Collaboration* and *Interaction*. *interactions+* represents a binary relation and has the form: *interactions: Collaboration→* ***some*** *Interaction.*

.

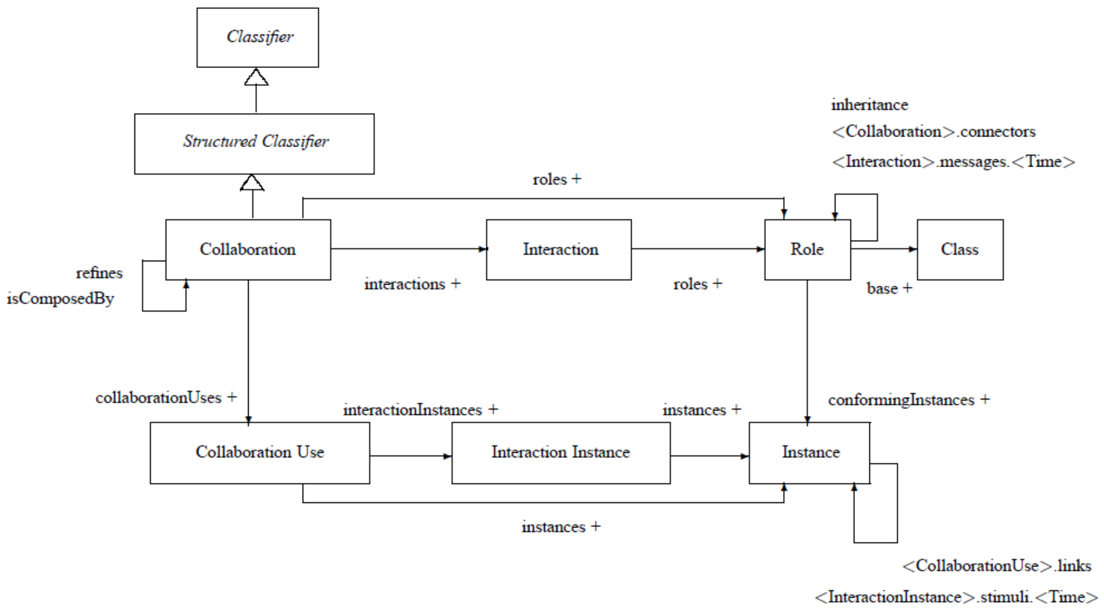


Fig. 5: A metamodel for collaborations based on the Alloy notation (part I)

This relation can be interpreted as every collaboration has one or more interactions. An arrow can represent one or more relations and it must have one label for each relation; for example the arrow from the signature Role to itself represents three relations.

Ternary relations can also be defined in Alloy; one of the relations attached to the *Role* signature is *<Collaboration>.connectors*. This relation is written using the Alloy notation for representing ternary relations; without using this notation this relation would be written as *connectors: Collaboration → Role → Role*.

Multiplicity symbols can be assigned to relations to constrain the number of instances participating in a relation; for example the multiplicity symbol + was added to relation roles to indicate that in a collaboration there must be a set of entities playing one of more roles. As explained in section 2, the default multiplicity in an Alloy diagram is any number. From fig. 6 and the relation inheritance of the *Classifier* signature, it can be concluded that a classifier can have any number of predecessors and that classifier can have any number of descendants.

Abstract signatures are specified in a diagram using an *italic font*; for example, the signature *Classifier* was specified as an abstract signature. As the reader can observe in figs. 5 and 6, only basic constraints can be defined in a graphical model. It is necessary to add further constraints with a textual model; this will be presented in the following section.

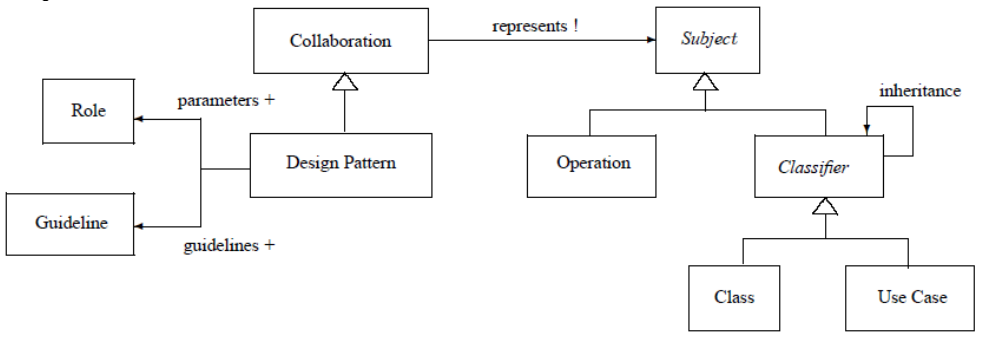


Fig. 6: A metamodel for collaborations based on the Alloy notation (part II)

5.0 THE FORMAL MODEL

Conventions used in the paper: In the model presented in this section, for the purpose of clarity, the keywords of the modeling language are printed in a **bold font** and model parts will be written in an *italic font*.

5.1 The Initial Model

Fig. 7 presents an initial model for collaborations based on the graphical models of figs. 5 and 6. It begins by defining the name of the model (*collaborationsModel*) in line 1 and the directory where this model is saved (*umlFormalModels*), then the signature *Collaboration* is defined. This signature defines that it is not necessary to assign a name to a collaboration (see line 3); the reason for this is because if a collaboration realizes a use case, a method or a class, the name for the collaboration is taken from these subjects [14]. By using the multiplicity keyword **some**, the second attribute of the collaboration signature declares that one or more entities with different roles participate in a collaboration. In line 5 the attribute *connectors* is defined; this defines a mapping from roles to roles; the keyword **set** in this line specifies that an entity with a given role can collaborate with zero or more entities with different roles. Line 6 defines that the dynamic part of a collaboration can be specified by using one or more interactions. As can be noted by the reader, multiplicity keywords permit the definition of some basic constraints on the model. More interesting constraints can be defined as appended facts, which are facts that constrain a specific signature and are defined near the constrained signature. For example, in line 8 an appended fact has been declared; this specifies that the number of connectors for a collaboration must be greater than two. The operator # returns the number of elements in a set.

```
1   module umlFormalModels/collaborationsModel

2   sig Collaboration{
3     name: lone Name,
4     roles: some Role,
5     connectors: roles set -> set roles,

6     interactions: some Interaction}
7     {
8     #connectors > 2
9     }

10  sig Role, Interaction{}

11  fact noIsolated{
12    all r: Role | r in Collaboration.roles
13    }

14  showInstance:
15    run{} for 3 but 1 Collaboration, 4 Role
```

Fig. 7: Initial model for the collaborations

Facts that apply to several signatures can also be defined. In line 11, a fact constraining entity which plays a role that must be assigned to a collaboration is defined; in other words, an entity must collaborate with other entities inside a collaboration.

The last line of fig. 7 is the **run** command and it is used to generate instances of the model; *showInstance* is a label that identifies this command. The number to the right of the keyword specifies the number of instances that can be generated from each signature. By using the keyword **but**, it is possible to specify an upper limit for the number of instances generated of one signature; for example, the keyword **but** specifies that the number of instances of the *Role* signature will be up to 4.

Fig. 8 presents an instance of the initial model generated with the Alloy analyzer. As can be observed, this instance represents a collaboration with roles 0, 1, 2 and 3. This instance allows one to detect two design errors of the initial model: 1. the entity that plays role 3 is not collaborating with any of the other entities and 2. the entities that play roles 1 and 2 collaborate with themselves.

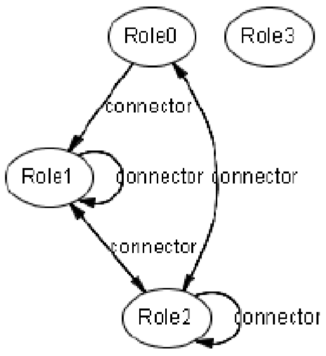


Fig. 8: Instance generated with the initial model

It is necessary to add some constraints to the initial model to eliminate these design errors; these constraints are shown in fig. 9. In particular, line 9 solves the first problem and line 11 solves the second problem.
Line 11 declares that the relation *connectors* between roles is irreflexive. This constraint has been defined so that if another relation presents this property, the only thing that needs to be done is to call the predicate *irreflexive*; so we can conclude that a predicate is useful in Alloy to make a section of the model reusable.
After the constraints of fig. 9 were added to the initial model, another instance was generated using the Alloy analyzer; this instance is presented in fig. 10. From this figure, it can be seen that the design errors contained in the initial model have been solved.

```
pred irreflexive[r: univ -> univ]{
no iden & r
}

sig Collaboration{
-- as before

}
{
-- as before

all r: roles | (r in connectors.roles)
or (r in roles.connectors)

irreflexive[connectors]
}
```

Fig. 9: Constraints added to the initial model

The following step defines some attributes of the Interaction signature and adds further constraints on the relation between this signature and the Collaboration signature, which are shown in fig. 11. As can be seen in this figure, an interaction is defined by a set of entities (line 3) (playing some roles) that interchange messages through time (line 6).

A message models the communication between two entities that interact in some time *t*. To understand this definition, let us define m as: *m=(roles* ***set→set*** *roles)*, then line 6 is converted to: messages: *m* ***set→set*** *times*; this indicates that a pair of entities that play roles r_1 and r_2, can communicate at several points in time as the second multiplicity keyword **set** implies. The first multiplicity keyword **set** declares that at a time *t* there can be zero or more pairs of entities that communicate with each other; this models that at the beginning of the interaction (t_0) no pairs of entities communicate.

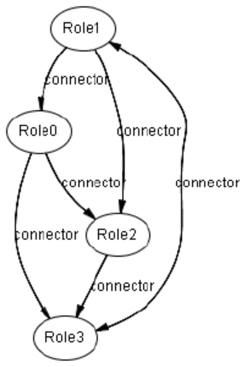


Fig. 10: Instance generated after some constraints were added to the initial model

Next, some appended facts are defined in the Interaction signature. Line 8 constrains that at least one message must occur in an interaction. The **let** statement allows for specification of a piece of model that is reusable inside a block; for example, in line 9 *m* is defined as *messages.Time* and it is used four times inside the appended fact. The following constraints are defined inside the appended fact:
1. All entities participating in an interaction must either receive or send a message (line 10).
2. An entity may not interact with itself in a given time *t* (line 12)
3. There is at least one collaboration in which the set of pairs of entities that interact in an interaction is a subset of the set of pairs of entities that work together in a collaboration (line 13).

```
1.  sig Time{}

2.  sig Interaction{
3.  roles: some Role,
4.  times: some Time,
5.  messages:
6.  (roles set -> set roles) set -> set times}
7.  {
8.  some messages

9.  let m = messages.Time{
10.   all r: roles |
11.   (r in m.roles) or (r in roles.m)

12.    irreflexive[m]

13.        some c: Collaboration | m in c.connectors}

14. sig Collaboration{
15.   --as before

16. }
17. {
18.   --as before

19.   some interactions

20.   let m = messages.Time |
21.     all i: interactions |
22.   (i.roles in roles) and (i.m in connectors)
23. }
```

Fig. 11: Enhancing the relationship between the Interaction and Collaboration signatures

The last constraints defined in fig. 11 are appended facts of the *Collaboration* signature, these are:

1. The dynamic part of a collaboration can be specified using one or more interactions (see line 21)
2. All roles defined in a collaboration must be defined in the interactions; if a pair of entities playing the roles r_1 and r_2 send messages in an interaction, then there must be a connector between these two entities in a collaboration (see line 22).

This subsection has explained the initial model for collaborations; in the following sections some improvements on the model will be made.

5.2 Modeling Operations on the Collaboration Model

So far some of the building blocks that form a collaboration have been defined (collaborations, interactions, roles and time), and the relation between these entities has been specified. Some constraints on these building blocks and their relations were also defined.

Alloy also has the ability to model operations on the models; an operation is defined as a predicate. Once an operation is defined, an analysis of the effects of this operation on the state of the model can be carried out by generating several model instances. .

Fig. 12 defines the way that participants in an interaction interchange messages over time. The predicate *init* defines one constraint on the initial state of the interaction; at the beginning of any interaction no participants interchange messages. The predicate *finish* states that at the end of any interaction, every participant p_1 of an interaction must send or receive one message to or from another participant p_2.

The predicate *addMessage* is a more interesting predicate; it defines what happens between the first and the last state of an interaction; it receives three parameters: the interaction that is going to be processed, the variable t that represents the time before the operation modifies the state of the interaction, and the variable t' that represents the time after the operation modifies the state of the interaction.

```
1.  open util/ordering [Time] as T

2.  pred antisymmetric[r: univ -> univ]{
            no ~r & r
3.  }

4.  pred init[t: Time]{
5.    all i: Interaction | no i.messages.t
    }

6.  pred addMessage[i: Interaction, t, t′: Time]{
            i.messages.t in i.messages.t′

7.    let m = (i.messages.t′ - i.messages.t){
8.      antisymmetric[m]
9.          some m
10.   }
11. }

12. pred finish[t: Time]{
13.   all i: Interaction, r: i.roles |
14.     r in i.messages.t.Role or
15.     r in i.messages.t[Role]
16.  }

17. fact DefineInteractions{
18.   init[ T/first[] ]

19.   all i: Interaction, t: i.times - T/last[] |
20.     let t′ = T/next[t] |
21.     addMessage[i, t, t′]

22.     finish[ T/last[] ]
23.  }

24. showInstance: run{} for 3
25. but 1 Collaboration, 4 Role, 1 Interaction,
26.     5 Time
```

Fig. 12: Traces

Lines 9 and 10 define two post-conditions; line 9 declares that all messages between the participants that occur at a time *t* must also occur at time *t*′; in other words, the messages at time *t*′ are composed of the messages at previous time *t* and the messages that occur at time *t*′. Line 10 defines *m* as the messages that occur at time *t*′; then lines 11 and 12 define some constraints on *m*. Line 11 in particular declares that it cannot be possible that at time *t*′, entity e_1 with role r_1 receives a message from entity e_2 with role r_2 and, at the same time, entity e_1 responds to entity e_2. Line 12 specifies that at a time *t*′, there must be at least one new message between two entities.

Once the desired behavior has been defined for the beginning, the end and during the interaction, the next step is to generate a simulation to analyze the effect of the *addMessage* operation on a collaboration state. Fig. 13 presents a simulation generated by the Alloy analyzer. As can be seen, this simulation fulfills all of the properties stated in the fig. 12. This completes the analysis of the operation *addMessage* on the collaboration model. The rest of the paper will enhance the initial model by adding more signatures and relations.

5.3 Adding Information to Roles

Fig. 14 focuses on roles; it adds some signatures to the initial model that are necessary for modeling the roles (*Attribute, Operation, Class and Instance*) and also adds some attributes and constraints to the *Role* signature. A role has one or more classes as its base (line 7). The operations and attributes of a role must be a subset of the operations and attributes of the class that acts as its base; this is specified in lines 15 and 21. In line 15 the keyword **this** represents the instance of the signature *Role* that the function *getClassifiers* takes as a parameter. Some interesting new features of the Alloy language appear in line 21; in this line a function, which is a reusable piece of model that can return a value, is defined. In this case the function *getClassifier* returns the set of classes that form the base for a role; this set is specified inside the *getClassifier* using the set-builder notation. A conforming instance of a role is an instance that has all the attributes and operations of that role. In other words if an instance wants to play a role, the former must have all the operations and attributes of the latter (see lines 8, 16 and 25).

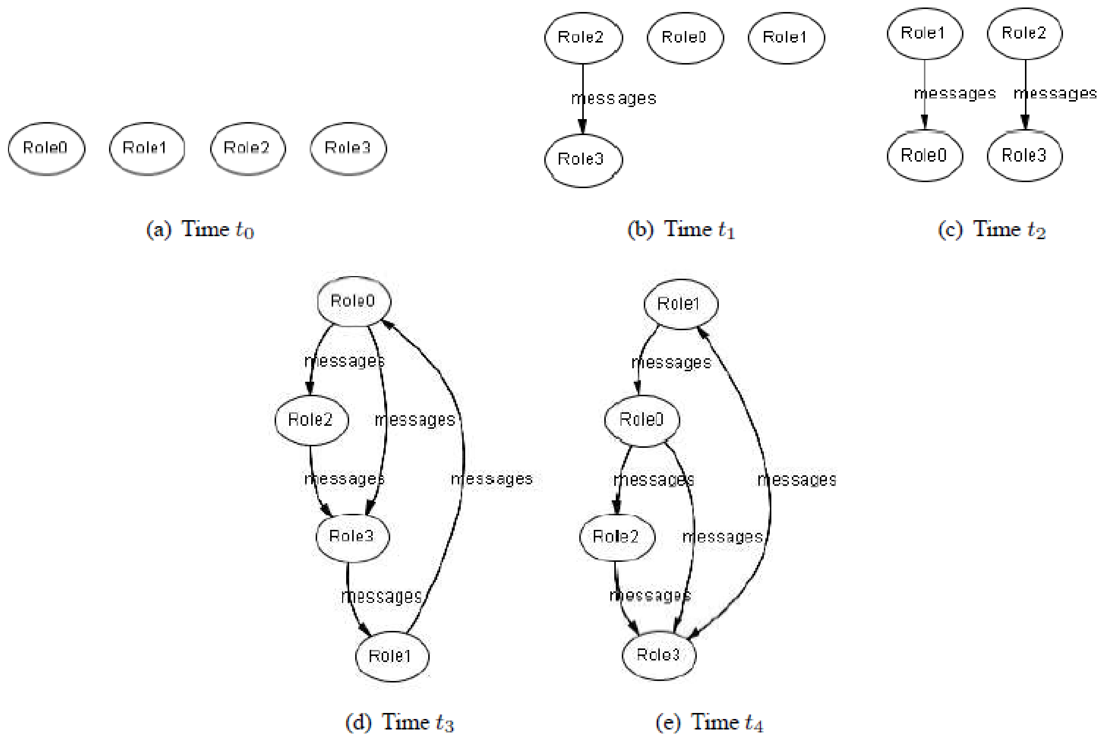


Fig. 13: Interaction *i* of the Collaboration *c* at different Times *t*

The multiplicity attribute defines the number of conforming instances of role *r* (see line 9).
The signature *Role* defines the relation inheritance between roles (see line 13); all the roles derived from one role must contain the attributes and operations of that role (line 18).

In line 34, a constraint that relates roles with classes, is defined; every role is a view of one class, which means that the attributes and operations of one role is a subset of the attributes and operations of one class.

The last constraint of figure 14 states that every instance must have the same attributes and operations of one class (see line 44).

```
1    sig Attribute, Operation{}

2    sig Class{
3      attrs: some Attribute,
4      ops: some Operation
5    }

6    sig Role{
7      base: some Class,

8      conformingInstances: some Instance,

9      multiplicity: Int,

10     attrs: some Attribute,
11     ops: some Operation,
12     availableFeatures: attrs + ops,

13     inheritances: set Role }
14   {
15     base = getClassifiers[this]

16     conformingInstances = getInstances[this]
```

```
   #conformingInstances = multiplicity

   all c: inheritances |
   (attrs + ops) in c.(@attrs + @ops)
 }

fun getClassifiers[r: Role]: set Class{
 { c: Class |
  r.(attrs + ops) in c.(attrs + ops) }
 }

 fun getInstances[r: Role]: set Instance{
 { i: Instance |
    r.(attrs + ops) in i.(attrs + ops) }
 }

 sig Collaboration{
 -- as before

 }
 {
   --as before

   all r: roles | some c: Class |
      isAViewOf[r, c]
 }

 pred isAViewOf[r: Role, c: Class]{
   r.(attrs + ops) in c.(attrs + ops)
 }

 sig Instance{
  attrs: some Attribute,
  ops: some Operation}
 {
   some c: Class |
   (attrs + ops) = c.(attrs + ops)
 }
```

Fig. 14: Definition of roles

5.4 Collaboration Uses and Interaction Instances

As was explained in the introduction, a collaboration can be used to model a class, a use case or an operation; these represent the attribute subject of a collaboration and will be explained later.

A particular collaboration could be used to model more than one of these subjects. Every time a collaboration is used to realize a subject a collaboration use is defined. A collaboration use can be defined as an instance of a collaboration; in fact in the document generated by the OMG of the UML 1.5, collaboration uses were called collaboration instances [14].

```
sig Collaboration{
  -- as before

  collaborationUses: some CollaborationUse}
  {
  -- as before

  all cu: collaborationUses, i: cu.instances |
    some r: roles | conforms[i, r]

  all cu: collaborationUses, r: roles |
    some i: cu.instances | conforms[i, r]

  all cu: collaborationUses |
   some r1,r2: roles, i1, i2: cu.instances{
    (r1->r2 in connectors && i1->i2 in cu.links)
    (conforms[i1, r1] && conforms[i2, r2])
    }
15     }

16   pred conforms[i: Instance, r: Role]{
17     r.(attrs + ops) in i.(attrs + ops)
18     }

19   sig CollaborationUse{
20     instances: some Instance,
21     links: instances set -> set instances,

22     interactionInstances:
23       some InteractionInstance }
24     {
25     some links

26     let m = stimuli.Time{

27       all i: interactionInstances |
28         i.instances in instances &&
29         i.m  in links
30       }

31     }

32   sig InteractionInstance{
33     instances: some Instance,
34     times: some Time,

35     stimuli:
36      (instances set -> set instances) set ->
37        set times
38     }
```

Fig. 15: Collaboration use and Interaction instance

Fig. 15 is focused on the definition of collaboration uses and interaction instances. An interaction instance models the dynamic part of a collaboration use.

The constraints defined in the signature Collaboration establish a link between collaborations and collaboration uses.

These constraints are explained as follow:

- A collaboration use defines a link between every role defined in a collaboration and every instance defined in a subject. Each of these instances must conform to a role; which means that the instances must possess all the features defined in a role (line 6).
- Every role defined in a collaboration must be played by at least one instance defined in a collaboration use (line 8).
- If two instances i_1 and i_2 that respectively conform roles r_1 and r_2 collaborate in a collaboration use, then there must be a connector between r_1 and r_2 in the based collaboration of the collaboration use (line 10).

A collaboration use is composed of the set of instances that conform the roles of a particular collaboration. In a collaboration use the entities communicate with each other using the links as a channel for the stimulus. A link conforms to a connector of one collaboration. A link is perhaps more restricted than a connector. A collaboration use defines the set of instances that conform the roles stated in a collaboration, as well as the links between these instances. An interaction instance defines how these instances collaborate together. All the instances defined in an interaction instance must also exist in a collaboration use (see line 28). If two instances communicate in an interaction instance, then there must be a link between these instances in a collaboration use (see line 29).

5.5 Relations between Collaborations

In this section the relations between collaborations are defined. Fig. 16 presents an UML example that considers all of these relations. In this figure, collaboration 2 extends collaboration 1, collaboration 3 refines collaboration 1 and collaborations 1, 2 and 3 compose collaboration 4.

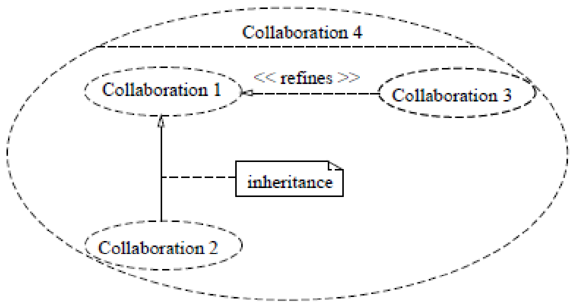


Fig. 16: Relations between collaborations

Fig. 17 contains some signatures and constraints added to the formal model to consider the relations between collaborations. The following relations are considered in the model: generalization (see line 8), refinement and composition (see line 14).

Using the *inheritances* relation, one collaboration can extend another collaboration. The reader should note that the *inheritances* relation of collaborations was not defined as an attribute of the *Collaboration* signature; instead this attribute was defined in the *Classifier* signature and then inherited to the *StructuredClassifier* which was later inherited to the *Collaboration* signature.

Two constraints applied to the inheritances relation: all the roles defined in a parent collaboration must also be defined in a child collaboration (see line 17) and the inheritances relation is permitted only between classifiers of the same type; in this case collaborations (see line 18).

A collaboration can also be defined in terms of other collaborations using the *isComposedBy* relation [20]; this allows the modeling of nested collaborations. In fig. 16 collaboration 4 is composed by collaborations 1, 2 and 3. A collaboration may refine other collaboration to give more details; this can be specified in a diagram using the refinement relation. Some constraints on the refinement relation must be added to the model to avoid the following inconsistencies:

1. If a collaboration c_1 refines a collaboration c_2, then c_2 may not refine c_1 (antisymmetric property).

2. Collaboration c cannot refine itself (irreflexive property). In line 20, the necessary constraints to avoid these inconsistencies were defined. These constraints were defined using predicates, which are reusable constraints.

In the model, these constraints were not specified for *inheritances* and *isComposedBy* relations, but they also apply to these relations.

```
1    pred acyclic[r: univ -> univ]{
2      no iden & ^r
3      }

4    sig Name{}

5    abstract sig Classifier{
6      name, package: lone Name,
7      interactions: set Interaction,

8      inheritances: set Classifier
9      }

10   sig abstract StructuredClassifier
11     extends Classifier{}

12   sig Collaboration extends StructuredClassifier{
13     -- as before
```

```
  refines, isComposedBy: set Collaboration }
  {
  -- as before

  all c: inheritances | roles in c.@roles

  inheritances in Collaboration
  }

fact ConstraintsRefines{
  antisymmetric[refines]
  irreflexive[refines]
  acyclic[refines]
  }
```

Fig. 17: Relations between collaborations

A collaboration can be used to model different levels of abstraction. It can be used to model methods, classes and use cases [17]; these are called subjects. The following section specifies how a subject can be realized by a collaboration.

5.6 Subjects Realized by Collaborations

Figs. 18 and 19 model subjects that can be realized by a collaboration. Some appended facts that define constraints on the subjects are defined inside the Collaboration signature. In particular, line 7 declares that a collaboration cannot be the subject of another collaboration.

5.6.1 Realization of Methods

Figs. 18 and 19 show the necessary signatures and constraints to model the realization of a method using collaborations.

According to [2], there are three forms for realizing a method: the use of code, an activity diagram or collaborations. If the method is too simple it can be specified using code in some programming language or it can be explicitly visualized in a note, but if the method is algorithmically intensive, the best choice for specifying the method is by using an activity diagram.
If the method presents a set of entities that work together to carry out the behavior of the method, then the correct way to realize the method is by using a collaboration. The parameters of the method, the return value, the local variables of the method and the variables contained in the classifier containing the method are the participants of the collaboration; they constitute the structural part of the collaboration.

All these participants must conform to a role defined in the collaboration (see line 9).

```
1   abstract sig Subject{}

2   sig Collaboration extends StructuredClassifier{
3     -- as before

4     represents: Subject }
5     {
6     -- as before

7     this not in represents

8     represents in Operation implies{
9      all c:
10       represents.(arguments + localVariables ) |
11        some r: roles | isAViewOf[r, c]

12     represents not in roles.ops
13      }

14    represents in Class implies{
15      no r: roles |
16        isAViewOf[r, represents & Class]

17      all c:
18       let arg = arguments, lv = localVariables,
19         go = globalObjects |
20          represents.( ops.(arg + lv) + go ) |
21            some r: roles | isAViewOf[r, c]
22      }

23    represents in UseCase implies
24      all c: represents.elements | some r: roles |
25        isAViewOf[r, c]

26    }
```

Fig. 18: Modeling subjects realized by collaborations (part I)

When a collaboration represents an operation, a role defined in the collaboration cannot be contained in this operation (see line 12).

The elements defined inside the signature *Method* are the elements that participate in realizing the method; these elements were defined so that there is no intersection within these elements, as the keyword **disj** implies (see line 30).

5.6.2 Realization of Classes

RUP is an iterative process, which incrementally builds the software (see [13]). At the beginning of this process the classes constituting the system are identified and only the external properties are defined. In the design stage, the internal design is defined for each class.

A collaboration can be used to specify the way the attributes of a class interact with the global objects accessible to that class and the parameters defined in the methods of the class. Line 15 specifies that if the subject to be specified is a class then none of the roles of the collaboration is a view of this class.

In line 18 a more complex constraint is defined: for all the participants collaborating in a class (arguments and local variables) there must be at least one role defined in a collaboration so that this role is a view for a participant.

5.6.3 Realization of Use Cases

RUP is a use case driven software development process; this means that all steps in the development process rely on use cases. A use case represents a requirement or restriction that the system must fulfill. Once the use cases of the system have been identified, the designer must to change these use cases into a representation that contains the entities that work together to obtain the functionality of the use cases. In other words, the designer needs to realize the use cases using collaborations. Every use case must be realized by one or more collaborations. Most of the time the refinement of use cases is not shown in a use case diagram; so it is important for a tool that specifies UML models to have a link between use cases and the collaborations that realize them. A use case can be realized by a collaboration identifying the necessary elements to carry out the use case (see line 46).

```
27    sig Operation extends Subject{
28    name: Name,

29    classes: some Class,
30    disj arguments, localVariables,
31    returnValue: classes }
32    {
33    classes = arguments + localVariables
34    }

35    sig Classifier extends Subject{

36      --as before

37    }
```

```
38  sig Class extends Classifier{
39    --as before
40  globalObjects: set Class}
41  {
42  all o: ops | this not in o.arguments
43  this not in globalObjects
44  }
45  sig UseCase extends Classifier{
46  elements: some Class
47  }
```

Fig. 19: Modeling subjects realized by collaborations (part II)

These elements can be discovered using a process like the one proposed in [22]. For each of these elements there must be a role defined in the collaboration so that this role is a view of one element (line 23).
Each of the scenarios of the use case can be realized with an interaction diagram.

5.7 Design Patterns

Fig. 20 models design patterns as types of collaborations. A collaboration does not represent a design pattern (see line 6), because the later is a type of the former.

```
1   sig Collaboration extends StructuredClassifier{
2     --as before
3   }
4   {
5     --as before
6   no represents & DesignPattern
7   }
8   sig Guideline{}
9   sig DesignPattern extends Collaboration{
10  parameters: some Role,
11  guidelines: some Guideline}
12  {
13  some name
14  parameters = roles
15  }
```

Fig. 20: Design patterns modeled as collaborations

When a collaboration realizes a design pattern it is necessary to assign a name to the collaboration (see line 13). All parameters of the design pattern must play a role in the collaboration (see line 14).

From the last two subsections it is concluded that collaborations represent the realization of use cases, operations, classes and that a design pattern is a type of collaboration.

6.0 CONCLUSIONS

In this paper the formal specification for collaboration was given. Alloy, which is a formal method and a modeling language, was used to specify collaboration models. A graphical model of collaborations was made to have an overview of a system and a textual model was constructed as a complement of the graphical model. The rules that govern how the elements in a collaboration can be connected were defined using these two models. The effect of the operations on the state of the system can be analyzed with the Alloy analyzer, as it was demonstrated in the analysis of the *addMessage* operation. In the personal opinion of this author, Alloy is an excellent tool for specifying collaborations because its simple but powerful notation. The specification made in this paper can be used as a guide for building a collaboration modeling tool.

REFERENCES

[1] J. Araujo and A. Moreira. "Specifying the behavior of UML collaborations using association for information systems". *Americas Conference on Information, Systems*, 2000.
[2] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley Professional, 2nd edition, 2005.
[3] M. Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, 3rd edition, 2003.
[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, 1995.
[5] A. L. Guennec, G. Suny, and J.-M. Jzquel. "Precise modeling of design patterns". *In Proceedings of UML 2000; LNCS*, 1939:482–496, 2000.
[6] K. Hamilton and R. Miles. Learning UML 2.0. O'Reilly, 2006.
[7] B. Hnatkowska, Z. Huzar, and L. Tuzinkiewicz. "Refinement of UML collaborations". *Int. J. Appl. Math. Comput. Sci.*, 16(1):155–164, 2006.
[8] D. Jackson. "A comparison of object modeling notations: Alloy, UML and Z". 1999.
[9] D. Jackson. Alloy: "A lightweight object modeling notation". *ACM Transactions on Software Engineering and Methodology*, 2002.
[10] D. Jackson. Software Abstractions: Logic, Language and Analysis. MIT Press, England, 2006.
[11] D. Jackson and M. Vaziri. "Finding bugs with a constraint solver". *In Proc. International Conference on Software Testing and Analysis*, Portland, Oregon, United States, 2000.
[12] S. Kendall. Fast Track UML 2.0. Apress, 2004.
[13] P. Kroll and P. Kruchten. The Rational Unified Process Made Easy. Addison Wesley, USA, 2003.
[14] OMG. UML 1.5. 2003.
[15] OMG. UML 2.0 superstructure specification. 2005.
[16] G. Overgaard. "A formal approach to collaborations in the unified modeling language". *The second international conference on the Unified Modeling Language; LNCS* 1723, 1999.
[17] T. Pender. UML Bible. John Wiley & Sons, 2003.
[18] B. Pilone and N. Pitman. UML 2.0 in a Nutshell. O'Reilly, 2005.
[19] T. Reenskaug. "The babyuml discipline of programming". *Software and System Modeling*, 2006.
[20] J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley Professional, 2nd edition, 2004.
[21] I. Sommerville. Software Engineering. Prentice Hall, USA, 8th edition, 2006.
[22] A. Weitzenfeld. Ingeniería de Software. Thomson, México, 2004.

BIOGRAPHY

Fernando Valles-Barajas obtained a graduate degree in Computer Science at Center for Research and Graduate Programs of La Laguna Institute of Technology (1991).
He received an MS in Control Engineering (1997) and a PhD in Artificial Intelligence (2001) from Monterrey Institute of Technology (ITESM) campus Monterrey.

He was a research assistant at Mechatronics department of ITESM campus Monterrey (1997-2001).

He received certification as a PSP Developer from the Software Engineering Institute of Carnegie Mellon University (2008).

He is member of the IEEE and ACM.

His research interests include topics in Software Engineering and Control Engineering.

Currently he is full-time professor in the Information Technology Department at Universidad Regiomontana, Monterrey, Nuevo León, México. He also teaches modules at both BS and MS levels in Computer Science and Software Engineering.