Constraints in Object-Oriented Analysis

Stefan Van Baelen*, Johan Lewi, Eric Steegmans and Bart Swennen

Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Leuven (Heverlee) - BELGIUM
Tel: +32 16-20.10.15 x 3596 - Fax: +32 16-20.53.08
E-mail: Stefan.VanBaelen@cs.kuleuven.ac.be

* Research Assistant of the Belgian National Fund for Scientific Research

Abstract

Object-oriented analysis methods can incorporate the concept of constraints to express rules of the problem domain in the specification model, restricting the possible instances of the model. As such, constraints describe properties that must be true at each moment in time for the entire system, without determining how they are to be preserved. The ways in which these constraints are introduced in the model differ from method to method, and even between distinct constraint types in a single method. Different ways in which constraints can be described, are illustrated and compared.

Specifying constraints as informal annotations or by operational restrictions is too informal and low level for analysis. According to the properties, importance and influence of the constraint types on the object model, they ought to be described differently. Some constraints, such as connectivity constraints, should best be integrated in existing model concepts to focus on the constraint during the concept definition and as a reminder for these kind of constraints Others, such as attribute value constraints, are best introduced as independent items part of a separate concept grafted on a general model to get a consistent, unambiguous, symmetrical and general applicable constraint description. Yet others, such as relational and existential dependency constraints, should be expressed implicitly by a hierarchical model structure. This approach enriches the object model in such a way that it highlights the logical structure of the problem domain to its right extent.

Keywords

Object-Oriented Analysis, Constraints, Object Model Structure

1. Introduction

Many object-oriented analysis methods have incorporated the concept of constraints to reflect rules of the external world in the specification model. Constraints are a means to express general properties of objects of a class. Constraints must hold during the entire life time of all objects of a class. As such, the number of possible valid instances of the specified model is diminished because the information present in the system must obey the constraint rules. By means of constraints, intrinsic properties of the system to be modeled can be described in a very elegant way. Indeed, in formulating constraints at the analysis level, only the aspect of '*what properties must be satisfied by objects of the class*' is covered, thereby abstracting from how these properties can be and when these properties must be controlled. These aspects are deferred to the design phase of the software life cycle.

The ways in which these constraints are introduced in the model differ from method to method. Constraint rules can be incorporated in the structure of the model, they can be introduced as a distinct concept next to the more classical ones, they can be integrated in the specification of the existing concepts, they can be described in an operational way or they can be considered as a mostly informal addendum to the model specification. The existing object-oriented analysis methods use a mixture of different specification techniques for distinct constraint types. However, some ways not always reflect the importance of certain constraint types, while some others are improper notations that cannot be applied consistently to related cases. The different ways in which constraints can be described will be examined and compared.

2. Constraints in O-O Analysis

Constraints play an important role in most object-oriented analysis methods. Their importance in the method is mostly reflected in the specification notation, which can vary from an informal textual part of the description to integrated or separate description parts. The description of constraints is almost totally neglected in some analysis methods, while other methods use them as glue to compose the whole model together properly. None of these extremes are desirable. We will discuss the different ways for constraints, highlight their advantages and disadvantages and suggest the domains in which they can be used.

2.1 Constraints as Informal Text

Almost all current object-oriented analysis methods have only informal support for general constraints on the model structure. However, some special constraint types are supported more formally. The degree in support for constraints varies from method to method. Some object-oriented analysis methods, such as OOA [Coad & Yourdon 91], Responsibility-Driven Design [Wirfs-Brock *et al.* 90] and OOSA [Shlaer & Mellor 88], neglect almost totally the importance of constraints. Properties of the external world cannot be expressed explicitly, but only as an additional part of the OOA documentation set. Instead of incorporating them as a major part in the analysis phase, they are considered to be a minor point of interest of the model. This leads to a neglect of the important role of constraint in the external world and in the analysis model of this external world. Although constraints are generally of utmost importance, whether they express rules of logic, physics or human-defined law and regulations, they would never get the same impact on the developed software models.

This is part of one of the current controversies in object-oriented analysis, namely formal versus informal specifications. One of the main reasons stated for informal specifications is to stimulate the creative process of analysis by not imposing strict rules on this analysis process. As a consequence, a strict formal description is also rejected. We think that, although the analysis process must keep its creativity, its outcomes must be formal. Indeed, if the design phase must start with an informal analysis description, errors will almost certainly be inevitable. An informal specification is exposed to human interpretation. This will not often correspond with the intention of whoever formulated it. Too much is left to the interpretation of the designer. If a formal analysis description with clear, well-defined semantics is produced, the following phases of the software life cycle have a solid base to go on with the development of the software system. In addition, formal verification techniques can be used to verify the obtained analysis model before going into design. This will prevent logical development errors from the analysis phase on. Also, although the outcomes of the analysis process must not be executable, it will be a good thing to make them interpretable. Then,

efficient checking, testing and prototyping can be done at the analysis level, and maybe even a straightforward implementation can be produced semi-automatically. Therefore, we try to provide next to a formal description of the effect of the actions, a formal definition of the constraints for the problem domain.

We will try to illustrate this with a simple example which we will use throughout the text. Consider a Banking system in which banks grant car loans to their clients. A person can only get a car loan if she/he is client at the bank (she/he owns an account), if she/he has reached the legally defined adult age and if she/he is of course using the money to buy a car (it will be considered as the mortgage for the loan). A possible model for such system is presented in Fig 1.

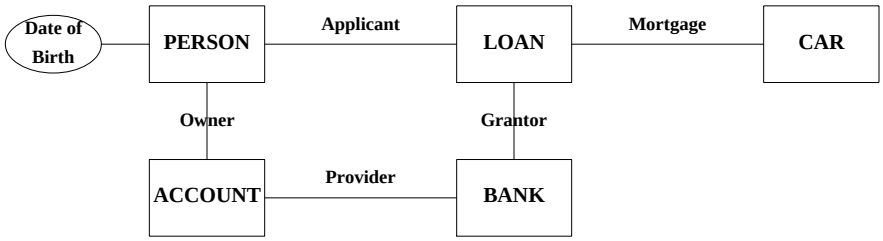

Fig. 1

The informal constraints that have to be added to this model are the following :

A. An account can only exist if a person and bank are related with it.
B. A loan can only exist if a person, a bank and a car are related with it.
C. The person related with the loan must be of adult age.
D. The person related with the loan must have an account at the same bank that granted the loan.

These informal specified constraints will never get the same important impact on the analysis outcomes as the original constraints have on the external world. More formal specification techniques are more fitted to express the importance of these constraints. Constraints are, just like classes, attributes, relations and actions, important items of an object-oriented analysis model. Therefore, the analysis model must contain a more formal definition of constraints instead of the informal textual constraint definitions. Different ways in which this can be done are presented in the next sections.

2.2 Constraints as Operational Restrictions

One possibility to incorporate constraints in the actual model is by controlling the execution of actions. This is supported by almost all current object-oriented analysis methods. By means of state transition diagrams, control flow diagrams or pre- and postconditions on actions, violations of the real world properties can be avoided. Such approach causes several problems.

A first problem is the gap that is introduced between the problem space and the obtained analysis model. Instead of describing what rules apply in the real world, the analysis model describes how they are realized. This must certainly be specified during the development, but at the design level rather than at the analysis level. The analysis phase must be centered around the reflection of the external world into the developed model, a direct mapping from the problem domain to the system model. Specifying constraints by means of action control

introduces a gap between the problem domain and the system model. The final goal of a workable software development method is not only a smooth transition from analysis through design to implementation, but also from the problem space to the analysis model. It is actually this transition that gets disturbed when constraints are not an important concept of the analysis method, but realized by artificial, not well-suited means.

Another problem of specifying the constraint realization instead of the constraint itself concerns future revisions and modifications. When a constraint is specified as an independent item, it will remain present as such in revised and modified systems. However, when the constraint is realized in an operational way, it would be hard to see the consequences of new additions or changes to the model on the constraint realization. The revisor must perform a sort of reverse engineering by extracting the constraint from its realization. Therefore, it is much better to specify the constraint as a model item than to realize it by restricting the existing model actions to enforce the constraint. You can compare it with a class invariant in some programming languages. Because the invariant will remain applicable after the addition of new actions, it is easy to keep the system consistent by preserving the invariant. However, if this invariant is only realized by means of pre- and postconditions on actions, it would be very hard to keep consistency and see the consequences of adding new actions.

To realize the previous described constraints by controlling the execution of actions, almost every constructor and mutator of all present classes will be influenced. Indeed, every class in the model of Fig. 1 can contain a mutator that causes a violation of a specified constraint. This is because every class has the right to change the instances of the relations in which it takes part. The possible (semi-) control flow diagrams for constraints A, B, and D are represented in Fig. 2. It only expresses the order of construction and destruction of the objects. The interpretation, for instance of the diagram for constraint A, is as follows : A creation of an object of *ACCOUNT* is only possible after the creation of objects of both *PERSON* and *BANK* which are not yet destroyed. Hereafter the destruction of that account object must precede the destruction of either the person or the bank object. So it is possible to create a new account immediately after the destruction of an account with the same participant objects.

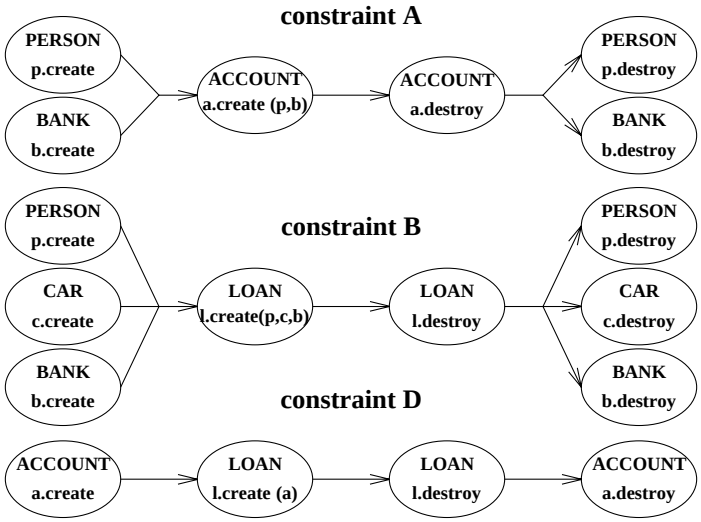

Fig. 2

If there are actions that can change the relations directly, such as moving an account from one person to another or changing the mortgage to another car, the state diagrams have to be extended. After an addition of a new action of any class, one is obliged to review the entire state diagram to keep its correctness. Indeed, because it is already a realization of the constraints, this realization has to be corrected after the slightest change of the model. Therefore, it is very hard to keep the constraint correct during the whole software life cycle, which contains many revisions, modifications and adaptations.

The constraint C can be expressed by means of a precondition on the creation of the objects of *LOAN*, for instance as follows :

```
class LOAN is
  constructors
    create ( p : PERSON, ...)
      require : p.age[1] ≥ adult_age
  ...
end LOAN
```

However, if new actions are added to manipulate the relation *Applicant*, these actions have also to be placed under control. Thus, every addition of an action causes a possible violation of the constraint. When the constraint was formulated directly, this would not be the case. Future additions will then not give rise to revisions of existing constraints.

The analysis phase is centered around the problem space, describing what must be done. Therefore, constraints must be specified as such, and not already be realized by means of operational constructs such as preconditions and control flow and state transition diagrams. The constraint specification must be formal and explicit to get the greatest benefit during the whole software life cycle.

2.3 Constraints Integrated in Existing Model Concepts

Most Object-Oriented Analysis Methods that try to incorporate constraints in their software model in a formal and explicit way, integrate them with other concepts of the method, mostly restricted to one model item of one concept. Constraints about relations and attributes are integrated in the definition of the relation and attribute, constraints about objects of a class are specified as part of the class description and so on. This can be very useful and adequate for certain groups of constraints. Other constraint groups however are defined for one of the existing concepts or model items, although they can spread out over several of them[2].

The definition of connectivity (multiplicity) constraints on relations is almost always integrated in the relation definition. Obviously, such constraint is a basic part of a relation. Separating the connectivity from the relation definition will introduce the possible danger of

---

1. Actually, we can get the age of a person by comparing the current time with the date of birth. This can be expressed as '**now** - p.Date of Birth'. Because the age of a person is a semantic real world property of a person, we will define it as a query *age* for the class *PERSON*, returning the age at the actual moment in time.
2. There is a slight difference between constraints that spread out over more than one concept and constraints that spread out over more than one item of the same concept. An example of the first type is constraint C, that includes both an attribute and a relation, whereas constraint A, B and D only deal with relations between classes, but more than one at the same time.

overlooking this important aspect of relations during the development of the analysis model. Another example of a useful integration is the multiplicity of attributes (one or many possible attribute values for an object) and the mutability properties of an attribute. It is useful to specify if certain attributes may only be defined at creation time of the object or can change during the life cycle of the object. For instance, the date of birth of a person may only be defined at the birth of a person and it may not more be changed during the life of that person. Also the name of a person[3] , the agreement date of an account and the approval date of a loan are examples of immutable attributes. On the other hand, the address of a person, his length and weight are examples of attributes of which the actual value will vary during the life cycle of a person. The absence of a mutator for the attribute is not enough because these mutator can always be added later without any control. So the property of immutability has to be defined as such. These constraints that can be integrated easily, only bear upon a single item of one concept of the analysis method, which is in the previous cases one relation or one attribute definition.

However, if a constraint can spread out over more items of the same concept or several concepts, it is not possible to integrate constraints with a particular concept in a decent manner. Constraints over several relations cannot be placed with one particular relation definition. A method may decide to place rules between attributes of the relation participants in the relation definition, e.g., as in OMT [Rumbaugh *et al.* 91]. But, for instance, rules between attributes of objects connected by one or more consecutive relations, or join and anti-join constraints in a relation ring (if an object *a* of class A is connected through successive relations with *a*', also of class A, then a = a', respectively a ≠ a') cannot be adjudged to a particular dedicated relation or class. Such constraints spread out over the whole model instead of some instances of just one concept. When such constraints are placed in a single class or with a single relation, arbitrariness will have a huge impact on the model. Indeed, we could have chosen an alternative viewpoint for placing and specifying the same constraint. In some cases, it would be even hard to see that two constraints are actually identical. Also, the information distribution in the obtained model will be very asymmetrical. Useful information for classes is hidden in the definition of other classes, relations or attributes. Another possibility, next to placing constraints in one particular class, is to place a constraint in every class that is influenced by it. This will give rise to an enormous amount of information duplication and overloading. Bad placement of constraints in the model will also lead to the diminishing of reuse. It will be very hard to get a proper insight in the existing structure, which will give rise to start all over again. A separate notation mechanism for constraints influencing more than one specific element of the model is therefore advisable.

The example of the previous section can be extended with connectivity constraints, as presented in Fig. 3. An account and a loan can be related with at most one person, one bank and one car. Persons and banks can be related with many accounts and loans. However, a car can be related with at most one loan, because it serves as the mortgage for the loan. The multiplicity of the attributes will also cause no problems for the model. It is obvious that a person can only have one date of birth at each moment in time.

The only problem that arises is the allocation of the four constraints of our example to the best fitted model items already present. Constraint A can be placed with the class *ACCOUNT*, but also with the two relations *Owner* and *Provider* (This is often done by means of the notion of a mandatory participant in a relation). Constraint B is of the same kind as constraint A. Constraint C can be placed with the classes *PERSON* or *LOAN*, with the relation *Applicant* or with the attribute *Date of Birth*. The choice between these alternatives will always remain

3. Neglecting the legal possibilities to change one's name

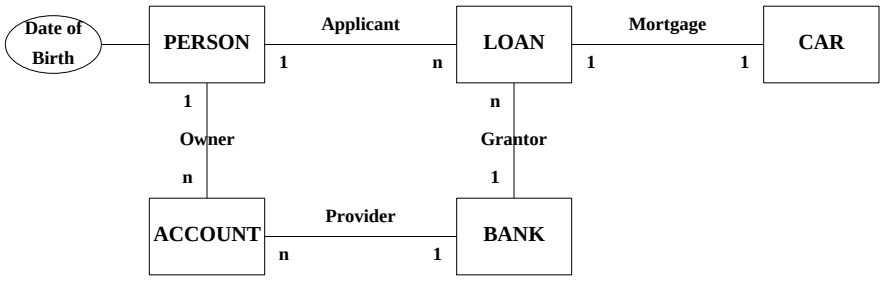

Fig. 3

rather arbitrary. Constraint D is an example of a join constraint in a relation ring. There are four classes and four relations involved. Which criteria can we use to select one of them instead of the others? No matter which one we choose, it will always introduce arbitrariness and asymmetry in the model.

To conclude, we can say that some constraint types are indeed strong related to existing concepts. An example of them is the connectivity (multiplicity) restriction for a relation. It would be obvious to integrate them with the concepts they belong to. However, most constraints spread out over a number of elements of the model and can therefore not be placed properly with only one of them. Therefore, we would like a mechanism to specify constraints formally and explicitly, but next to the classical model concepts that are present in the existing analysis methods.

2.4 Constraints as a Separate Concept

To overcome the difficulties of placing constraints with the existing concepts of the analysis model and to get a workable specification mechanism for all constraint types, a separate notation mechanism for constraints is developed as part of the ERO OS project[4] [Lewi *et al.* 90, Van Baelen *et al.* 92, Lewi *et al.* 93]. Such constraint specifications contain the actual specification of the constraint itself in first-order logic, a name as a mnemonic for it, and the part of the model on which it is applied. This affected model part is automatically derived from the actual specification of the constraint and is especially useful for design purposes (see further). The specification of the involved classes can include the constraint name to highlight the restriction on their instances. However, the constraint definition will be a separate information item of the model, not integrated in other model elements. Otherwise, some of the problems of the previous case would appear again. The notation of constraints as a separate concept leads to a consistent, unambiguous and symmetrical constraint specification and placement.

We will illustrate this with a specification of some of the previous mentioned constraints in Fig. 4. We will use the same names for the constraints as given earlier.[5]

---

4. ERO OS is an object-oriented development method that supports the full software life cycle, from analysis (ERO OS-A), through design (ERO OS-D) and implementation (ERO OS-I), to the maintenance and running (evolution) phase. A full description of the ERO OS method falls beyond the scope of this paper.

5. The operation ‘a.Relation Name’ stands for the set of objects to which the object *a* is related by the relation ‘Relation Name’. The required adult age is modeled as a predefined value of the domain *DURATION*, which is an abstract data type for indicating a period in time.

```
constraint A
 --    for ACCOUNT with relations
 --              Owner, Provider is
 for each a in ACCOUNT :
   a.Owner ≠ empty set
    and a.Provider ≠ empty set
end A
```

```
constraint B
 --    for LOAN with relations
 --      Applicant, Grantor, Mortgage is
 for each l in LOAN :
    l.Applicant ≠ empty set
     and l.Grantor ≠ empty set
     and l.Mortgage ≠ empty set
end B
```

```
constraint C
 --    for LOAN with relations Applicant
 --         to PERSON with attributes Date of Birth is
 for each l in LOAN :
    l.Applicant.age
          ≥ DURATION'adult_age
end C
```

```
constraint D
 --    for LOAN with relations
 --      ( Applicant to PERSON
 --          with relations Owner,
 --        Grantor to BANK
 --          with relations Provider ) is
 for each l in LOAN :
    l.Applicant.Owner intersection
     l.Grantor.Provider ≠ empty set
end D
```

Fig. 4

The only asymmetry that is still present arises from the fact that the affected model part and the constraint specification is described starting from one of the involved classes. However, this affects only the specification of the constraint itself, not the position of the constraint in the whole model. In the next section, we will abolish this asymmetry totally by introducing a hierarchy between classes and relations. We can then specify every constraint starting from the highest class(es) of the hierarchy.

The constraint specification can be extended with the specification of a trigger, a sort of exception handling mechanism for the constraint. The trigger describes a number of actions that must be performed when a violation of the constraint occur. In this case, the trigger actions are executed and hereafter the constraint must become valid again. The rule that a constraint is always satisfied is still valid. Only the actions that are used to keep the constraint valid are stated precisely. This trigger constraint is especially useful for the specification of constraints involving time, when certain actions must be performed after the exceeding of a predefined moment in time. The action that causes the violation of the constraint, the evolution of time, cannot be prevented.

The cases that are not well supported by this constraint specification technique are twofold. In the first place, classes that are actually materializations of relations, such as *ACCOUNT* (relation between *PERSON* and *BANK*) and *LOAN* (relation between *PERSON*, *BANK* and *CAR*), always imply an additional constraint on the presence of an object of the participant classes. Examples of them are constraint A and B. They emerge due to the fact that a relationship cannot exist without the knowledge of the participants it relates. As such, an account cannot exist without the person and the bank that are part of the account relationship. When the relation *ACCOUNT* is transformed into a class, the old characteristics must be enforced by means of a constraint.

The second inconvenience appears when one object is dependent of the existence of another object. Such dependency is expressed by constraint D, stating that a loan can only exist if a person has an account at the same bank. As such, there exist an existential dependency

between each loan object and a corresponding account object. Such constraints are like glue, holding the model instances consistent w.r.t. the problem space. When a flat model is developed, many structure dependent constraints have to be enforced explicitly. This approach is not favorable, neither of the viewpoint of the model developer nor of that of the model reviewer and re-user.

This is because the logic structure dependencies are only specified by means of additional constraints, not by the model structure elements themselves. The modeler has to make an explicit transition between the logic structure of the problem space information and the model equivalent, a combination of model structure elements and structure related constraints. The reviewer of a loose model with many structure constraints will have to put the pieces of the puzzle together before she/he gets insight in the model. Instead of highlighting the basic structure of the model, one of the important things of an analysis description, this structure is neglected and shifted to the specified constraints. Moreover, it is possible to end up with a certain model without having thought of the implied structure constraints that are present in the problem space. A good model should capture many constraints through its structure. This is however not the case in a flat relation structure.

To conclude, we can say that the notation of constraints as a separate concept leads to a consistent, unambiguous and symmetrical constraint specification and placement for all constraint types. But important structure dependencies are hidden in these constraint specifications, instead of being part of the basic model structure. Therefore, we will extend our basic model structure elements to reflect the logic structure dependencies directly in the model structure. Then, the model will capture the current structure and relation dependency constraints implicitly.

2.5 Constraints Implied by the Model Structure

To diminish the gap between the logical structure and the model structure by enriching the expressive power of the model structure elements, a hierarchical relation structure can be used. A hierarchical relation structure will treat relations as classes themselves.[6] As such, the choice to model a certain relational thing, e.g., accounts, as a relation or a class will disappear. It will be modeled as both class and relation. Such class will be called a refined class (or a class refined by a relation), because the relation refines the objects of the class as relationships between objects of other classes.[7] Such approach has two major benefits concerning constraints : It captures implicitly the structure constraints and the constraints about the relationship objects. By defining the class *ACCOUNT* as a class refined by a relation between *PERSON* and *BANK*, it is stated that account objects also represent a relationship between a person and a bank. Therefore, such objects cannot exist without being related to a person and a

---

6. OMT [Rumbaugh *et al.* 91] and OSA [Embley *et al.* 92] also provides the possibility to model an association as a class. But it is not really a class of objects with their own identity, but merely a class of associations of two objects, which identities are defined as a combination of the identities of the objects part of the association. Duplicates, for instance, are not possible. Furthermore, you have to make a choice already at the analysis level whether a problem space relation is going to be modeled as a straight relation, a relation as a class (actually an exceptional case in OMT, rarely used), or just a class with two additional relations for the two participants. In ER©DS-A, every relation is encapsulated in a class. It is only in the design phase that we have to decide if a relation will be implemented by means of a class or an ordinary association.

7. The definition of an attribute for a class is called a decoration of that class. One can see a class as a naked body, that can be dressed with all kind of information : refined by a certain relation, decorated by several attributes and so on.

bank. This property is generally referred to as existential dependency. Constraint A has thus become superfluous.

Existential dependencies among objects may seem too restrictive for the ultimate system. A great deal of run-time flexibility, for instance in populating the model with instances, would be lost. However, object-oriented analysis is basically concerned with building an abstraction of the external world. Therefore, focusing on the external world in its *normal* appearance should have priority over the run-time issues. Refining accounts thus means that, under normal circumstances, each account must have an owner and a provider.

Before treating the dependencies for *LOAN*, the structure constraint D will be examined. This constraint can be reformulated as follows : When a person wants to get a loan from a bank, she/he has to own an account at that bank. So a loan can only exist if an account is associated with it. Therefore we refine the class *LOAN* by a relation between *CAR* and *ACCOUNT*. The account part of a loan captures the applicant and the grantor of the loan, together with the account that must exist before the loan can exist. The account serves as the contract base for the negotiated loan. Finally, this relation for *LOAN* also captures the dependency constraint B. So the only constraint that needs to be formulated explicitly is constraint C. The final model is presented in Fig. 5.

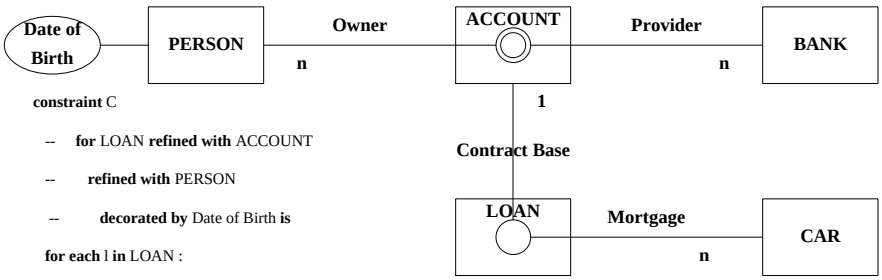

Fig. 5

The query 'a↓Role Name' (e.g., l↓Contract Base) in the constraint specification stands for the object of the Role Name class that is a basic part of the relationship object *a*. The inverse query 'b↑CLASS NAME' stands for the set of objects of the CLASS NAME class in which *b* participates. If $a \simeq (b,c)$[8] then a↓Role B = b, a↓Role C = c, $a \in$ b↑A and $a \in$ c↑A. This notation supports the view of zooming into the elements that are part of an object (projection ↓), and zooming out to the items that contains the object (election ↑). The circle for *LOAN* represents the relation (between *ACCOUNT* and *CAR*) that is encapsulated in the class *LOAN*. The double circle for *ACCOUNT* indicates that duplicates are allowed for the encapsulated relation. So a person can have more than one account at the same bank. Duplicates are indeed possible when relationships are considered as objects. Indeed, the object identity will always present, even if two objects have the same participants.

8. a is not really equal to (b,c), but it contains (b,c). In addition, it has its own object identity, which allows also that another object of class A, say a' ≃ (b,c).

When a class is existential dependent of only one other class, this can be expressed as in Fig. 6. In this case, the class is refined by a unary relation of the participant class. It is also possible to define a connectivity value for unary relation to express the fact if a participant object can participate in one or many refined objects.

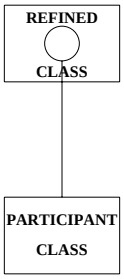

Fig. 6

Notice that we can now avoid arbitrary asymmetry in the constraint specifications. The description of the affected model part and the constraint itself will be seen from the highest classes in the relation hierarchy. As such, it is always definite how a certain constraint must be formulated. This mechanism is incorporated in ER0OS-A,. Connectivity and attribute multiplicity constraints are integrated with existing concepts, structure and relationship constraints are implied by the model structure whereas the remaining constraints are described as independent model items.

We can try to go even further and capture the remaining constraint also in our model structure by providing a special construct for it. For instance, by introducing subgroups within classes, we can define the group of adult accounts by defining the group of adults for class *PERSON*. This construction will capture also constraint C in its model structure, as presented in Fig. 7. The oblique line in the left corner is an indication of a subgroup of a class.

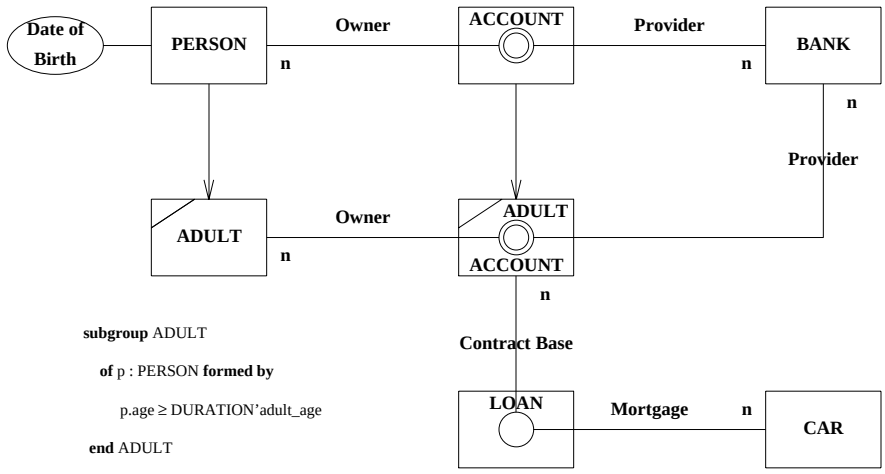

Fig. 7

However, a trade-off is necessary between model complexity and model simplicity and understandability. The step from a flat relation structure to a hierarchical relation structure has shown its use in practice. It provides a solid base for mapping the problem space into an analysis model. Whether or not more constraints have to be supported explicitly in the model structure, is not clear yet. But we are planning to experiment in the future with new structure mechanisms for capturing more constraints into the model structure. After evaluation of each proposal on criteria like '*model complexity versus model simplicity and understandability*', the accepted concepts will be added to the existing ER00S-A modeling concepts. We are also planning to extend our constraint specification mechanism with temporal logic to be able to express temporal constraints, such as '*this property until that happens*' or '*this property will become valid in the future*'.

A summary of the advantages and disadvantages of the discussed specification techniques is presented in Table 1.

Constraint specification	Advantages	Disadvantages
Informal text	Expressivity of English language	Neglection of constraints Imprecise description No verification possible
Operational restrictions	Classical technique	Operational description Gap between analysis and problem domain Revision problems
Integrated in other concepts	Logical description Focus on constraint in concept definition Specification reminder for this kind of constraints Useful for relation connectivity, Attribute multiplicity and mutability of attribute and relation participant	Arbitrariness and improper description when used over several items
Separated specification	Consistent, unambiguous, symmetrical and general applicable description	Not fitted for relation materializations and existential dependency No reflection of logical structure No specification reminder for these constraints
Structure implied	Model highlights logical structure	Change in logical structure causes alteration in model structure Too complex when applied in extreme

Table 1

3. Realization of Constraints during O-O Design

The analysis of a software system must be focused on the problem space. The final outcomes of the analysis phase contains, amongst others, a complete description of the entire system on a high level of abstraction. Aspects of the solution domain are not incorporated in the analysis model. Therefore, constraints are introduced as a high level specification mechanism of rules and regulations of the problem space, without incorporating the decision of how and when they are going to be checked and realized. The design phase will introduce more details about the actual realization of the specified model. This is the right place and time to take the decisions regarding to the realization of the specified constraints. This section will present some design issues regarding constraints.

The design approach presented here is aimed at classical object-oriented languages, such as Smalltalk, C++ and Eiffel. Further research about design issues that arise when constraint-based object-oriented languages are used as a target language is planned in the near future.

3.1 Design Issues for Separate Constraints

Constraints that are specified as a separate concept item, as described in a previous section, still have to be realized in the lower levels of the system development life cycle. At the design phase, several topics arise during the constraint realization. The main issues are concerned with the place and time the system must be checked for possible constraint violations, and the actions that must be performed when a violation is going to occur or has occurred. Two distinct approaches can be applied :

- The first approach consists of preventing the occurrence of a constraint violation. First of all, the set of actions that can be the source of a constraint violations are derived. For each action, a require condition can be determined that must be fulfilled before the action can be executed. These require conditions will prevent the system of going into a wrong state. The execution of certain actions cannot be tolerated for certain system states. This approach causes a loss of efficiency due to a high number of tests, but keeps the system in a highly consistent state at each moment in time.
- Another approach consists of detecting fault system states, whereupon the system itself will perform either a sort of rollback to an older valid state, or the invocation of an error recovery procedure that tries to fix the system. This results in an important gain of efficiency, but leaves the system in an inconsistent state during a certain time.

The choice between these two approaches is often situation specific. A trade-off has to be made between efficiency and consistency, depending on which criteria are of utmost importance for the required system.

A tunable constraint realization approach by preventing constraint violations is already developed as part of the ERCOS project. The amount of decisions that have to be made is ordered in several successive levels, for a separation of concern. We will locate where and when each constraint can be violated. This will reduce the moments and the objects to be checked. Reduction is done in four steps :

- Firstly, the classes that are involved in each constraint will be determined. The constraint has to be checked on each object of such class after the invocation of each of its actions. Other classes cannot be of any influence on the realization of the constraint. This will decrease the moments on which a constraint has to be checked.

- Then, the set of objects of the involved classes that have to be verified after an action on an object of the class will be determined. Mostly, it will not be necessary to check each object of the class, after execution of an action on a certain object of that class; checking the involved object will be sufficient. Such situations will give rise to a decrease of the objects on which a constraint has to be checked.
- Thirdly, the actions that can cause a constraint violation must be determined. Actions concerning certain characteristics of an object, whether they are attributes or participants, will mostly not violate constraints about other characteristics. The moments on which the constraint has to be checked will hereby decrease.
- Finally, the require conditions that have to be tested before each action can be invoked, are derived. Mostly, these require conditions can be made simple and efficient, compared to the actual constraint expression. These transformations from the original constraint specification into a rather straightforward tests will increase efficiency quite a lot.

3.2 Design Issues for Implied Constraints

A hierarchical structuring of relations may result in more classes and a more complicated structure to implement. Therefore it is often advisable to transform the introduced structures to a simpler structure, a flat one for example. It is rather straightforward to transform the developed hierarchical model into a bipartite, flat model, consisting of classes on the one hand and flat relations on the other. Flat relations are preferred at the design level for reasons of simplicity and implementation ease. There is no identity or functionality associated with a flat relation. A flat relation corresponds with the relation concept of entity-relationship modeling and many object-oriented methods. It is also possible to change a refined class together with its relation into a flat relation, with the consequence that the functionality of the class must be shifted to neighboring classes. In this way, the number of classes of the analysis level can be diminished at the design level. This design level is the right place to decide which classes have to be optimized and which ones have to be kept as such. The main concern here is to find a good balance between the information amount and the procedural amount of the design.

4. Conclusion

After comparison of the different ways to specify for constraints, our conclusion for integration in the ERODS Method is the following :

- Specifying constraints as informal text is too informal as an outcome of the analysis phase. This will give rise to the introduction of human interpretation errors during later stages of the development.
- Specifying constraints explicitly by operational restrictions is too low level during analysis. Such approaches are not advisable because they cause a gap between the problem space and the analysis model. Instead of describing what rules apply in the real world, the analysis model describes how they are realized. In addition, they have always to be converted from their conceptual meaning to their implementation and vice versa.
- Certain constraints, such as connectivity and attribute multiplicity constraints, and mutability constraints of attributes and relation participants can easily be integrated in existing model concepts. When constraints can spread out over several concepts, it is not advisable to define them for just one of them. This leads to asymmetry and arbitrariness in the constraint specification.

- Constraints can be considered as independent items part of a separate concept. It highlights the importance of constraints to its right extent. However, constraints about the relationship objects and structure constraints have to be enforced explicitly. Instead of highlighting the basic structure of the model, one of the important things of an analysis description, this structure is neglected and shifted to the specified constraints.
- A hierarchical relation structure will capture such constraints implicitly by the model structure. This will reduce the relationship and structure constraints, often present in flat models, and avoid arbitrary asymmetry in the constraint placement. It will highlight the logical structure of the model to its right extent.

No other special constructs are included yet in ER◊OS, as trade-off between model complexity and model simplicity and understandability. But we are planning to experiment in the future with new structure mechanisms for capturing more constraints into structure. Such structure mechanisms can be introduced in ER◊OS-A after a positive evaluation.

References

[Coad & Yourdon 91] Coad, P., and Yourdon, E., *Object-Oriented Analysis Second Edition*, Yourdon Press, Englewood Cliffs (New Jersey), 1991.

[Embley *et al.* 92] Embley, D.W., Kurtz, B.D., and Woodfield, S.N.,, *Object-Oriented Systems Analysis*, Yourdon Press, Englewood Cliffs (New Jersey), 1992.

[Lewi *et al.* 90] Lewi, J., Steegmans, E., and Van Baelen, S., EROOS : Entity-Relationship Object-Oriented Specifications, Department of Computer Science, K.U.Leuven, CW Report 111, 1990.

[Lewi *et al.* 93] Lewi, J., Steegmans, E., Dockx, J., Swennen, B., Van Baelen, S., and Van Riel, H., *Object-Oriented Software Development with ER◊OS: The Analysis Phase*, Department of Computer Science, K.U.Leuven, CW Report 169, 1993.

[Rumbaugh *et al.* 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs (New Jersey), 1991.

[Shlaer & Mellor 88] Shlaer, S., and Mellor, S. J., *Object-Oriented Systems Analysis : Modeling the World in Data*, Yourdon Press, Englewood Cliffs (New Jersey), 1988.

[Van Baelen *et al.* 92] Van Baelen, S, Lewi, J., Steegmans, E., and Van Riel, H., EROOS : An Entity-Relationship based Object-Oriented Specification Method, *Technology of Object-Oriented Languages and Systems TOOLS 7*, Heeg, G., Magnusson, B., and Meyer, B. (ed.), Prentice Hall, Hertsfordshire, UK, pp. 103-117.

[Wirfs-Brock *et al.* 90] Wirfs-Brock, R., Wilkerson, B., and Wiener, L., *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs (New Jersey), 1990.