Dynamic First-Class Relations for Knowledge-based Systems

Alejandro Sanchez
INRIA Sophia Antipolis, FR
Universidad Nacional de Cordoba, AR
alejandro.sanchez@sophia.inria.fr

Sabine Moisan
INRIA Sophia Antipolis, FR
sabine.moisan@sophia.inria.fr

Jean-Paul Rigault
INRIA Sophia Antipolis, FR
University of Nice Sophia Antipolis
jpr@polytech.unice.fr

Abstract

In the context of knowledge-based systems interacting with dynamic or even real time processes, experts need to express domain specific relations in a natural way. We thus propose an expert language and a C++ implementation of these relations as first-class objects. The relations may have associated constraints such as multiplicities or any predicate expressing a necessary condition. The focus of this paper is to cope with time evolution of objects and their relations and, in particular, with the possibility of temporary inconsistencies. C++ components have been implemented to represent relations and their management and first experiments on existing knowledge-based systems are underway.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]; I.2.4 [*Knowledge Representation Formalisms and Methods*]

General Terms relation, object-oriented language, knowledge-based system

Keywords first class relation, dynamic system, domain specific language

1. Motivation

We are working on a generic object-oriented component framework for knowledge-based systems (KBS) (Moisan 2008) supporting several kinds of reasoning tasks (classification, planning, resource allocation, video understanding, activity recognition...). A KBS is a software program basically composed of a knowledge base written by a domain expert and an engine that performs inference reasoning on this knowledge to solve a particular problem. The framework provides C++ components implementing basic concepts of KBS such as *frames* (sorts of classes), *slots* (structured attributes) and *daemons* (methods automatically triggered in case of slot or frame access), *inference rules*, and *inference engines*... It also allows the definition of Domain Specific Languages (e.g., (Moisan 2002)) that are used by experts to enter their knowledge. Once translated into C++ code, this expertise is linked with the engine to yield an executable KBS for solving (non expert) end users' problems.

These expert languages propose a syntax to describe application objects and to manipulate them through domain specific rules. Relations among objects constitute an indispensable part of the knowledge. These relations are intrinsic to the domain and may present various natures, e.g., spatial relations between objects in a scene, animals and their habitats, customer and service reservation... They seldom are functional or even reducible to a formal theory. Therefore, we need to provide experts with a direct means to assert that a relation holds. Moreover, our relations do not belong to the intrinsic properties of objects and thus should be separated from them. Embedding relations into object attributes does not favor this separation, nor is it a natural way for experts to declare and manipulate them. Hence, we decided to introduce relations as first class concepts in the expert languages as well as in the C++ components.

Two important properties of our KBSs are worth mentioning. First, experts and end users are not programmers, all the C++ code is automatically generated from expert knowledge, no customization is made at construction nor at execution time. Second, our systems are usually embedded and autonomous; they manipulate *dynamic* (even real time) objects. Thus the state of objects evolves with time during the KBS execution, which may challenge existing relation consistency. This paper concentrates on this topic.

This paper is organized as follows. In the next section we describe our model of relations. Then we give the flavor of the language proposed for experts to describe relations. Section 4 discusses various solutions to handle temporary inconsistent relations and details the C++ implementation. We terminate with a comparison with other works, the present status, and future work.

2. Proposed Relation Model

We chose to rely on the mathematical notion of a "relation", for the abstract definition as well as for the implementation. For the time being, we limit ourselves to binary relations. A binary relation $\mathcal{R}$ is a subset of the Cartesian product of two sets A and B: $\mathcal{R} \subseteq A \times B$. In our case the sets are composed of all the instances of a class (a.k.a, the class extension). A (resp. B) is the set that contains the binary relation domain (resp. codomain); for simplicity, we call A the *domain* and B the *codomain*. Hence, the relation $\mathcal{R}$ appears as a set of tuples (here, pairs) (a, b) where $a \in A$ and $b \in B$. Therefore, we reify the relations themselves, not the individual tuples.

In some cases, a collection of tuples is not sufficient. Thus, we also offer the expert the possibility to attach constraints to a relation, that is predicates that each tuple must satisfy. Note that these constraints add extra *necessary* conditions for a tuple to be part of a relation but they are *not*, in general, the characteristic function of the relation (which would be necessary and sufficient). The well known multiplicity constraints are of this kind, although they are usually expressed in a direct way. Other need a predicate involving the two related objects. For instance, in video understanding, two objects can be considered "close to" each other provided that they are both visible (not occluded) in the scene; for the relation

RAOOL October 2008, Nashville.

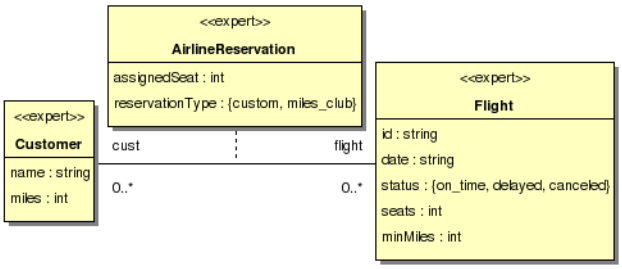

Figure 1. A simple example of relation

between customers and flights in airline reservation, two customers cannot reserve the same seat on the same flight...

There are two major forms of constraints: those which must always hold (they are structural invariants of the system) and those that may temporarily be violated provided that they hold at some system observation point. The first form is rather simple to handle: should such a constraints be violated, the system throws exceptions. The second form is more challenging, especially in real time applications. It imposes to correctly monitor the object state changes. Indeed, during KBS execution, object values evolve which may invalidate the constraints of some relation tuples. For instance, in process planning, the relations between objects depend on the initial conditions, the spontaneous evolution of the process, the effects of the previously planned actions, etc. Since evolutions cannot be predicted by the expert nor by the engine; we must provide an automatic mechanism to cope with relation inconsistencies (see 4.1).

Some objects or even complete applications, however, do not require relations and there is no reason to impose the corresponding cost on them. Therefore, we wish that domain objects depend as little as possible on relations so that they may be oblivious of relations. Of course, the reverse is not feasible: relations have to know the objects they relate but we shall keep this dependency as abstract as possible.

3. Expert's View of Relations

We extended the knowledge representation languages so that experts can define binary relations, specifying their domain and codomain. A relation as a whole may have its own attributes, multiplicities on both ends, and be associated with constraint predicates. These predicates express a (necessary) condition on a tuple, thus they take two arguments, one in the domain, the other in the codomain.

Here is an example of a possible expert's input. We consider the *AirlineReservation* relation between *Customer*s and *Flight*s, which is represented by an "association class" in the UML diagram of figure 1. Besides, the expert may set three constraints on this relation (among many possible ones): (1) it should not be possible to assign a non existing seat; (2) no reservation should exist over *"canceled"* flights; (3) if a customer has a *"miles club"* reservation, he/she must have enough miles for the reserved flight.

Assuming that the knowledge base already contains definitions of the *Customer* and *Flight* classes, the expert may define the relation as follows (keywords are in bold):

Relation
{
name AirlineReservation
domain Customer **ranges** [0,1] **role** cust
codomain Flight **ranges** [0,1] **role** flight

Attributes
Integer **name** assignedSeat
ReservationType **name** reservationMode

Constraints
seatExists **ensures that** assignedSeat <= flight.seats
notCancelled **ensures that** flight.status != canceled
enoughMiles **ensures that** reservationType == miles_club
implies cust.miles >= flight.miles
}

In the first three lines the expert declares the relation name, its domain and codomain and their multiplicity ranges. Each element is given a role name: we do not support a sophisticated notion of roles; role names are merely for easy naming of related objects in constraints. The next lines describe the attributes of the relation. Finally, the expert provides the three previous constraints. The syntax of constraints supports predicate logic (on finite collections). We presently impose no restriction on the predicate definition which can use objects and relation attributes as well as relation properties.

Then, the expert can directly manipulate *AirlineReservation*, mainly when writing decision rules. Possible operations are to add, remove, consult relation tuples, and extract all codomain objects related to a given domain one (and the converse). For instance, if a customer needs to travel somewhere, a standard rule could be

Let c **a** Customer, t **a** Town
If (c, t) **in** IsBoundTo
Then insert (c, t) **into** AirlineReservation [reservationType=custom]

where we assume that *IsBoundTo* is another relation, also defined by the expert. As shown, this rule both consults relation *IsBoundTo* and for each satisfying tuple, possibly inserts it into *AirlineReservation*; indeed, insertion may fail because of constraints. Note that relation attributes can be set when inserting, like for *reservationType* in the example.

Once complete, a knowledge base is automatically translated into C++. A relation type such as *AirlineReservation* becomes a C++ class which represents a collection of tuples, each tuple being a pair of (pointers to) objects. These classes automatically handle constraint verifications when a new tuple is inserted as well as when objects change their state (see 4.1). The resulting C++ code is linked with the engine and relation libraries, yielding an executable KBS that will be executed, without further expert input.

The advantage of an automatic code generation is that we may optimize the target code, e.g., avoid generating relation handling code for those classes not involved in any relation. On the negative side, experts cannot introduce new classes, relations, or rules at run time, which is not a real drawback for most of our target systems, which must run autonomously.

Localizing all the information about a relation in a unique class relieves the expert (and the generated code) from handling reference attributes in objects at both ends of the relation. Moreover, all decisions about object evolution follow up are centralized and can be easily tuned. In addition, operations on relations (inverting, combining...) would be easier to express in future versions. The experts adopt a natural declarative style: they simply define a relation together with its constraints and use it in rules; processing relations is handled transparently by automatically triggered functions (daemons in objects).

4. Framework Components for Relations

4.1 Coping with object state change

In a KBS, the object states may change due to the engine reasoning activity. In particular, related objects can change state independently and in any order. The consequence is that the constraints of some

relation tuples may become false, which invalidates the tuple itself. However, this invalidity may just be temporary or due to serialization of updates which are in fact logically concurrent.

For instance, in the reservation example, imagine that customer *John* needs to make a reservation on the *Nice_Paris* flight and can make it under the *miles club* mode, because he has enough miles. Suppose that he consumes too many of his miles for other trips before flying to Paris. Then, the relation tuple (*John*, *Nice_Paris*) becomes inconsistent since the third constraint is violated. However, it is still possible that *John* regains enough miles by flying to other destinations before honoring his *Nice_Paris* reservation. Until then, the reservation on the *Nice_Paris* flight is inconsistent.

Of course, these types of inconsistency do not concern constraints which represent structural invariants of the system, including multiplicity ranges (see 2): as soon has they are violated, an exception is thrown which may trigger "repair rules" defined in the knowledge base. In our KBSs, these repair rules allow to "backtrack", meaning that the system may forget state changes and resume in a previous (valid) configuration.

In the other cases, different solutions can be considered to cope with temporary inconsistencies:

1. Decide validity points where the relations are assumed to be consistent, meaning that in between they can be inconsistent and thus should not be consulted. In this line falls the use of *transactions* as used in data bases. In our case, this is not always practicable since these points must be decided either by the expert or by the engine. In some cases, the expert can express validity points using relation constraints, and all is well. For instance, it is easy to express that an airline reservation must be consistent two hours before flight departure. In many other cases, there is no meaningful information about validity points that can be drawn from the expertise. The only way would be an *ad hoc* modification of the generated code, something to avoid in our generative approach. Therefore, a default mechanism is needed for setting validity points. This cannot be done by the engine since it has no semantic knowledge about the objects and their relations in a particular KBS. Hence, it cannot choose semantically significant validity points. The only events it could track are low level, such as all object changes; checking relations at these points is not only expensive but it usually has no meaning and does not solve the problem of several objects evolving concurrently.
2. Check on demand: the constraints are verified only when using (accessing, consulting...) a relation tuple. This solution means that a relation must be consistent at each access. The main advantage of this approach is that the objects do not have to know about their relations. However, this solution may induce superfluous checks when no changes at all occurred or when the changes did not impact the constraints. Yet, this could be improved by simply marking the relation tuples when one member changes, but this breaks the independence of objects with respect to relations.
3. Use a temporary black list: when an object state changes so that some relation constraints are violated, put the inconsistent tuples in a "black list" and give a chance to related objects to change accordingly to satisfy the constraints again. In the latter case, the corresponding tuples go back into the relation. This allows to keep the memory of the relations stated by experts even when they are inconsistent and to cope with concurrently evolving objects. There are, however, two drawbacks. First, objects need to know in which relations they are involved. The second drawback is more problematic: how long should we keep the relation tuples in the black list?

We propose a compromise between solutions 2 and 3. When an object changes its state, we mark all relation tuples in which it is involved. When a relation is accessed, that is when a tuple is retrieved, we check this tuple only if it is marked. If the check succeeds, there is no problem and we can unmark the tuple. Otherwise, the relation does not hold for this tuple and it was up to the expert to specify one among several "remedial strategies" (globally for all relations or specifically for some):

S1 remove the inconsistent tuples; this comes down to solution in item 2 above (check on demand);

S2 blacklist inconsistent tuples until either they become consistent again (and thus are recycled as valid) or some expert rules decide to remove them;

S3 blacklist the inconsistent tuples until both objects are modified; then remove the inconsistent tuples and recycle the others.

In our reservation example, choosing remedial strategy **S2**, the actions on sequence diagram in figure 2 are triggered at the moment when *John*'s miles are modified (section 4.1. When *John* consumes miles, the reservation is notified and all tuples with *John* as *Customer* are marked. When *John* decides to query his reservation, the constraints of his marked tuples are checked. If they are satisfied, the tuple is unmarked and the query succeeds. Otherwise the tuple is blacklisted and, at the next query, a check will be performed again; if *John* has accumulated new miles in the meantime, the query can be successful.

Thus, when a relation or an object attribute (as in the example) is modified, all the relations to which the modified object belongs are notified. This solution implies that objects know their relations whereas we wish to keep objects as independent as possible of relations. Fortunately, this dependency can be kept to a minimum, similar to the dependency between an object and one of its observers. In fact the relation plays the role of an Observer for the objects.

4.2 Structure of the C++ Components

We give here a brief description of the C++ component design for implementing relations. As already mentioned, we reify a whole relation as a set of tuples (set of pairs in our case, since we restrict to binary relation). Figure 3 presents a simplified class diagram. The most important class is the template *Abstract-Relation*, which is parametrized by the types of the domain and the codomain and which contains the tuple set: each tuple is a pair of pointers (to domain and codomain objects) together with some application dependent information (e.g., the seat number of an airline reservation) represented by template parameter *Info*.

The relations declared by the expert are transformed into a class derived from *AbstractRelation* instantiated with the corresponding domain and codomain types. In our example, the *AirlineReservation* relation derives from *AbstractRelation<Customer, Flight>*. This class is generated from declarations given in expert's syntax in section 3. In particular, the three constraints of the example will be embedded by the generator into the *check* method of *AirlineReservation* that implements the pure virtual function *check* in *AbstractRelation*. This method is the one that will verify whether the relation constraints are satisfied. Hence, the base class defines all properties, attributes and methods common to all relations whereas expert subclasses add application specific information. The implementation of relation tuples provides the mechanisms to store, access, share, mark, and blacklist the tuples in a simple and efficient way.

As has been already discussed, application objects need to notify their changes to their relations. Thus each object has a reference to the set of relations in which it is involved. To reduce the

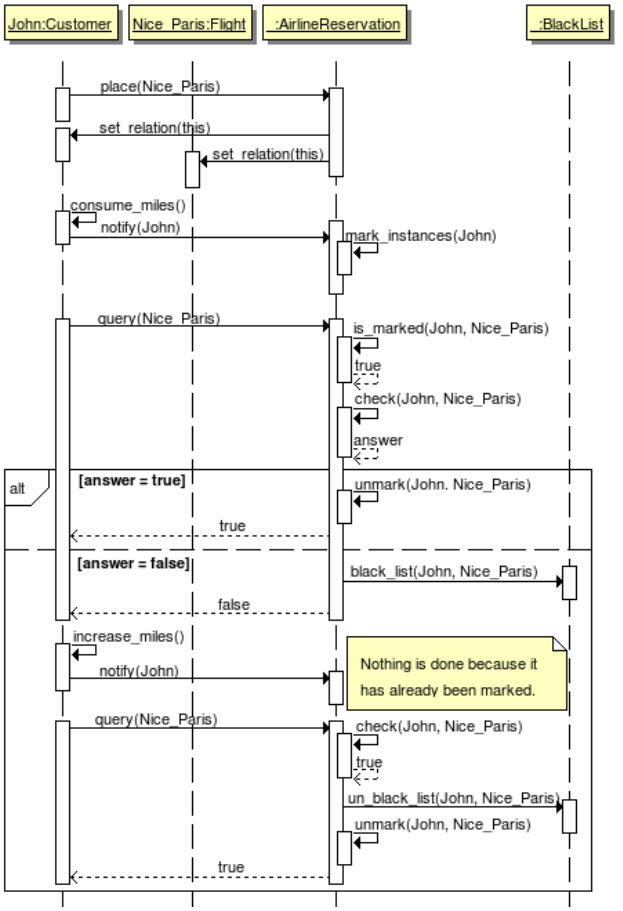

Figure 2. Management of temporary inconsistency (This scenario supposes that airline companies do not debit the miles account at reservation time but only at flying time, a really generous behavior!)

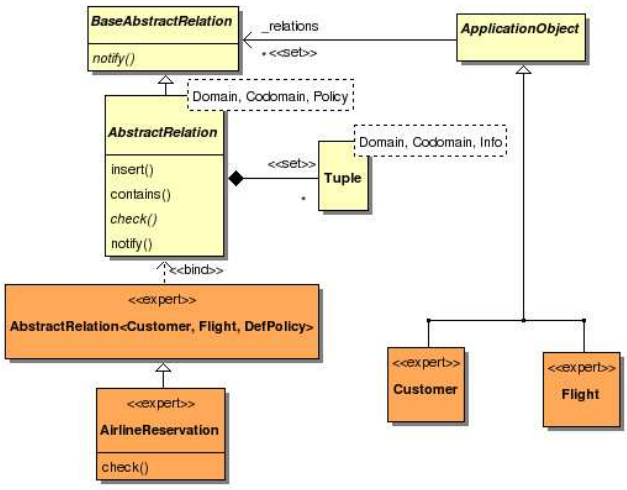

Figure 3. Structure of relation implementation in the KBS framework: classes in grey (or orange) are generated from expert definitions whereas all others are part of the framework.

object-relation dependency a non-template *BaseAbstractRelation* base class is used, which just provides an abstract interface through which real relations can be accessed.

The current C++ implementation relies on class templates and STL containers only.

5. Related Works

Most works aiming at relation support in object-oriented languages address the transformation of UML-like associations to code. Some implement associations as first-class objects (Page et al. 2001; Bierman and Wren 2005), other use different techniques such as patterns or aspects (Pearce and Noble 2006). Some have a notion of constraints or invariants attached to their relations (Page et al. 2001) (Génova et al. 2003). There is also an important line of work interested in connecting relations with the notion of roles (Boella and Steinmann 2008).

The goal is often to reason about relations or to verify the consistency of a relational systems or ontologies, for instance using some form of algebraic calculus (Balzer et al. 2007; Page et al. 2001). The target applications are data mining, information systems, knowledge base verification... In this case, relations correspond to structural properties of the objects.

By contrast, our goal is not to reason about pure structural relations but to address dynamic relations with constraints that can be temporarily unsatisfied. In our applications, objects change independently of their relations and relations themselves do not impact object evolution. Although previously cited works may address objects evolving with time, most of them do not take constraints or temporary inconsistencies into account. However, some authors have broached this problem, for instance (Génova et al. 2003) who suggest to delay constraint and multiplicity verification until accessing the relation. Unfortunately, they rejected an implementation of relations as first class, mainly for implementation reasons. For our part, we think that direct implementation of relations is not only natural but also practicable.

As far as implementation is concerned, there are many ways of representing relations (see for instance (Noble 1997) for a survey). Contrasting with Noble's "Relationship Object" our application objects do not point to the tuples in which they are involved, but to the relations that contain these tuples. By centralizing most of the information about the relation as a whole, so that we may experiment different design choices and strategies, as described in 4.1. We may also simply traverse the whole relation to perform some global operation (e.g., find all customers flying to Paris some time in the next week).

Several implementations are available either as extensions of object-oriented languages as Java (Bierman and Wren 2005) or as libraries in Tcl (Mangogna 2006), or using C++ templates (DOL) as we do. However, we are less concerned with persistence and we preferred a representation of the whole relation, mainly for experimentation purpose.

6. Present Status and Future Work

In this paper, we have presented an implementation of first-class binary relations for knowledge-based systems. These relations are associated with constraints which are dynamically checked. The possible temporary inconsistencies are handled through parametrized strategies. This is work in progress and we did not address here other problems such as n-ary relations, composition of relations or even relations between relations, that we are currently exploring.

At this time, we have defined an expert's language to describe relations together with its parser and C++ code generator. The generated code uses a set of C++ classes smoothly integrated into our KBS framework. Currently, we are "re-engineering" existing

knowledge bases to take advantage of relations and to evaluate the expression power and performance issues.

There are two major paths to introduce relations into an object-oriented language: extend the syntax and semantics of the language or implement a library. In fact we use both: we extended our domain specific language and the generated C++ code relies on a (template) library.

References

S. Balzer, T. R. Gross, and P. Eugster. A relational model of object collaborations and its use in reasoning about relationships. In E. Ernst, editor, *ECOOP 2007*, number 4609 in LNAI, pages 323–346. Springer Verlag, 2007.

G. Bierman and A. Wren. First-class relationships in an object-oriented language. In A. P. Black, editor, *ECOOP 2005*, volume 3586 of *LNCS*, pages 262–286. Springer Verlag, 2005.

G. Boella and F. Steinmann. Roles and relationships in object-oriented programming, multiagent systems and ontologies. In M. Cebulla, editor, *ECOOP 2007 Workshop Reader*, number 4906 in LNCS, pages 108–122. Springer Verlag, 2008.

DOL. *Data Object Library*. CodeFarms. http://www.codefarms.com.

G. Génova, C. Ruiz del Castillo, and J. Llorens. Mapping UML associations into Java code. *Journal of Object Technology*, 2(5):135–162, 2003.

A. Mangogna. TclRAL: A realational algebra for Tcl. In *13th Annual Tcl/Tk Conference*, Naperville, IL, October 2006.

S. Moisan. Knowledge representation for program reuse. In *European Conference on Artificial Intelligence (ECAI)*, pages 240–244, Lyon, France, July 2002.

S. Moisan. Component-based support for knowledge-based systems. In *ICEIS, 10th International Conference on Enterprise Information Systems*, Barcelona, Spain, June 2008.

J. Noble. Basic relationship patterns. In *In EuroPLOP Proceedings*. Addison-Wesley, 1997.

M. Page, J. Gensel, C. Capponi, C. Bruley, P. Genoud, D. Ziébelin, D. Bardou, and V. Dupierris. A new approach in object-based knowledge representation: The AROM system. In *Proc. 14th IEA/AIE*, number 2070 in LNCS, pages 113–118, Budapest, Hungary, 2001. Springer.

D. J. Pearce and James Noble. Relationship aspects. In *AOSD 06*, pages 75–85. ACM, 2006.