LNCS 2863, pp341-355, October 2003.
(URL: http://www.springerlink.com/link.asp?id=8ut6bbdlyfrvywf8)
Copyright © Springer-Verlag

Modeling and Testing Legacy Data Consistency Requirements

Jan Pettersen Nytun[1,2] and Christian S. Jensen[3,1]

[1] Faculty of Engineering, Agder University College
Grooseveien 36, N-4876 Grimstad, Norway
j.p.nytun@hia.no
[2] Department of Informatics, University of Oslo
P.O.Box 1080 Blindern, N-0316 Oslo, Norway
[3] Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark
csj@cs.auc.dk

Abstract. An increasing number of data sources are available on the Internet, many of which offer semantically overlapping data, but based on different schemas, or models. While it is often of interest to integrate such data sources, the lack of consistency among them makes this integration difficult. This paper addresses the need for new techniques that enable the modeling and consistency checking for legacy data sources. Specifically, the paper contributes to the development of a framework that enables consistency testing of data coming from different types of data sources. The vehicle is UML and its accompanying XMI. The paper presents techniques for modeling consistency requirements using OCL and other UML modeling elements: it studies how models that describe the required consistencies among instances of legacy models can be designed in standard UML tools that support XMI. The paper also considers the automatic checking of consistency in the context of one of the modeling techniques. The legacy model instances that are inputs to the consistency check must be represented in XMI.

1 Introduction

In addition to the Internet scenario mentioned in the abstract, when an enterprise adopts a new software system, that system must typically work in a setting with several existing legacy systems. It is often the case that new systems integrate and combine data and functionality from existing systems. In this paper, we focus on data integration. More specifically, our focus is on the consistency problems that occur when previously uncoordinated, but semantically overlapping data sources are being integrated.

For example, public administrations, in their strive to create one single, public IT infrastructure, might build a new system that integrates previously separate databases that concern different aspects of physical properties (land parcels, buildings, etc.). It is important to the new system that the different representations of the same physical properties be consistent. And the introduction of the new system provides an opportunity to improve the quality of the existing databases.

In this paper, we use UML (e.g., class diagrams) and its accompanying Object Constraint Language (OCL) to model and test for consistency in this setting. The paper explores different possible modeling techniques and forms a recommendation for how to accomplish consistency modeling and testing. The notion of *something being consistent with something else* can be applied in many contexts. For example, an implementation can be consistent with a model, meaning it is a correct implementation of the model. Our specific context is legacy systems: we consider how to model consistency among data managed by different legacy systems.

Most of today's legacy systems use relational technology for persistent data storage. Having two databases with overlapping models (i.e., parts of their schemas describe the same reality, or miniworld) one consistency rule could be: *two objects with the same identity, modeling the same real-world entity, must have the same values stored for corresponding attributes; otherwise, they are not consistent with each other.* We investigate how to exploit UML in this type of modeling situation and how to perform the actual consistency testing.

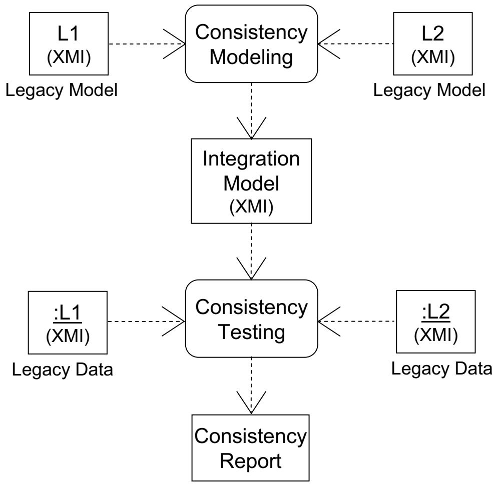


Fig. 1. Consistency Modeling Overview

Fig. 1 offers an overview of our approach. The models of the legacy systems are UML models represented in the XMI [1] format (metamodel level M1 [2]). The output of the *consistency modeling* is an *integration model* where the two legacy models have been integrated and the desired consistency has been expressed explicitly. We assume that the modeling activity is manual, but the paper's results might also be useful when designing automatic support for generation of integration models.

The paper explores the use of various subsets of UML for the consistency modeling, and it recommends the use of a particular subset of UML notation together with guidelines for this task. A tool or plug-in may be designed based on this, to give extra support for this modeling approach. However, an ordinary UML tool will do since the recommended notation is a subset of UML. Next, *consistency testing* is done automatically. The paper describes *consistency testing* tailored to one selected consistency modeling technique. The consistency model and legacy data are inputs to the *consistency testing* activity. The legacy data are instances of the legacy models that were integrated in the integration model. The data are represented according to the XMI format (metamodel level M0) and can, e.g., be snapshots of legacy databases. The output of the consistency testing activity is a report describing the consistency violations that were revealed.

Our approach is related to constraint programming [3]: the modeler declares constraints, and the test environment will later find a solution that is consistent with the constraints. We assign values to so-called consistency attributes by evaluating declared constraints. Our proposal also involves so-called consistency associations between legacy classes. For these, the test environment will, in a sense, try to break multiplicity constraints; and if it succeeds, a *wrong cardinality* consistency violation occurs.

Maintaining consistency among different representations of the same entity stored in different databases has been studied before [4], [5], [6]. We consider a notion of consistency modeling that seems more general than most related work. In comparison with the most related work [5], we do not rely on an extension of UML (we stay within UML), and our testing is quite different.

An extension to OCL has been proposed [7] with the objective of describing quality-ensuring constraints on geographic data. We do not extend OCL, but instead propose to introduce special associations and classes to support the specification of complex consistency constraints. We believe that this aids in obtaining a very practical approach.

This paper is structured as follows. Sect. 2 defines concepts, e.g., consistency model and integration model, that are used throughout the paper; a first example of consistency modeling is also introduced. In Sect. 3, different consistency modeling techniques are described. Sect. 4 proceeds to discuss the modeling techniques, and one technique is selected as the most useful. This section also covers the automatic testing of such a model. Finally, Sect. 5 offers a short discussion, conclusions, and directions for further work.

2 The Consistency Model

We proceed to define what we mean by consistency model, how it relates to Model-Driven Architecture (MDA) [8], and the role of the test environment.

2.1 The Test Environment

Consistency checking might be challenging in a highly dynamic context: OCL operators such as **forALL** and **allInstances** are hard to implement when objects come and go. The operators will function well if the object structure is static. In particular, a snapshot of a system is static and can easily be used (see [9] and [10] regarding the recording of distributed global state).

On the other hand, the restriction to a static context may impose limitations on the use of operation calls (query operations) in OCL expressions. If the operations are completely described in the model (e.g., by the use of action semantics) they might be interpreted at test time.

The test environment might offer some support for the consistency modeling. For example, in relation to geographic information, spatial functions may be part of the test environment. Such functions can then be used freely in OCL expressions. At consistency modeling time, the classes supported by the test environment can be seen as part of a special, dynamic legacy model (thus query operations will work properly). Such support can greatly strengthen the consistency testing.

2.2 An Initial Example

Consider a simple case that relates to object replication. The package IM shown in Fig. 2 contains legacy-model elements and additional modeling elements for describing consistency: IM is an example of an integration model, and two legacy models are shown that are stored in package L1 and L2, respectively. The package stereotyped ⟨⟨consistencymodel⟩⟩ is explained in the next section.

The dashed-dotted line between class L1::C1 and L2::C2 indicate that objects of type L1::C1 might be tested for consistency against objects of type L2::C2. The package notation is cumbersome to read, so we omit it in the remainder of the paper, even if this in a strict sense makes some expressions syntactically incorrect.

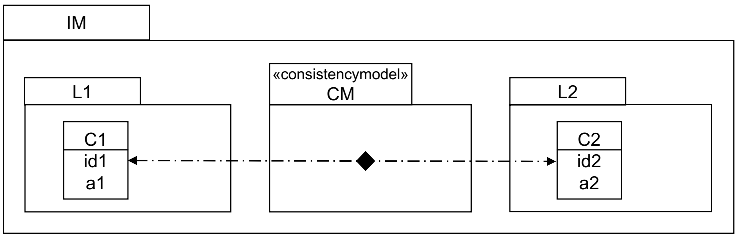


Fig. 2. Integration Model Encompassing Legacy Models and a Consistency Model

We should be able to capture the following consistency requirement in the integration model: if attribute values <u>:C1.id1</u> and <u>:C2.id2</u> are equal, it makes sense to talk about the consistency of objects <u>:C1</u> and <u>:C2</u>; they are consistent if and only if <u>:C1.a1</u> is equal to <u>:C2.a2</u>. Using OCL-syntax, we may state this as follows.

Context C1 **inv**:

C2.**allInstances**->**forAll**(self.id1 = id2 **implies** self.a1 = a2)

While it seems possible to do consistency modeling with OCL alone(at least with minor extensions [7], [11]) this will not benefit from the visual strengths of UML.

2.3 Consistency Model

We term the part of the integration model that is not part of any legacy model the *consistency model.* The package containing the consistency model is stereotyped ⟪*consistencymodel*⟫, as shown in Fig. 2.

With UML (XMI), we may put modeling elements into separate packages. It is for example possible to put the description of an association into a package separate from the packages of the connected classes. It is up to the modeler to store the consistency model in a separate, stereotyped package.

At *consistency test time*, an instance of the integration model is instantiated. Instances of legacy models are prefabricated and will be inserted as parts of the integration model instance. The test environment then automatically instantiates the consistency model. The consistency model can be seen as a declaration: instances of consistency model elements are in a sense derived from the legacy instances and the declaration.

Let us take a closer look at the OCL expression from the previous section, the core of which can be written as:

e1.id1 = e2.id2 **implies** e1.a1 = e2.a2

where e1 is of type C1 and e2 is of type C2. Conceptually, if e1 and e2 have the same id value, they must have the same attribute value to be consistent; and if their id values are different, consistency is not questioned. This expression is an instance of a more general pattern:

<match> **implies** <consistency test>

The first part of the expression defines *what to test* for consistency, and the last part defines *what is required for consistency* to hold. This separation seems sensible even if the specification of what to test may not be trivial: which attributes are fundamental (Aristotle's Law of Identity)? And what if the values of the identifying attributes are inconsistent? The full power of OCL can be used when defining the matching. Since OCL expressions can include operation-calls (query operations), advanced functional libraries can be applied if they are available in the test environment, e.g., spatial operations in a geographical system.

2.4 MDA and the Consistency Model

The notions of Platform-Independent Model (PIM) and Platform-Specific Model (PSM) are central to OMG's MDA initiative; a PIM is a model where (some) technical details have been abstracted away. A PIM might be mapped to a PSM, which is closer to implementation. In short, MDA advocates a software development approach where abstract models are mapped (manually or automatically) to less abstract ones until implementation is achieved. If a legacy system has been developed in accordance with MDA, there will be several models that are candidates for consistency modeling. In this case, a decision of which abstraction level to use for the consistency model has to be made.

For the consistency testing to be correct, the user data must be at the same level as the integration model. Both user data and the integration model might be subject to mapping to achieve this "compatibility." This type of mapping might not be trivial: assume for instance that the integration model consists of two PIM legacy models and a consistency model that contains at least one OCL constraint that references elements in both PIMs; further, assume that the two legacy systems has been implemented on different platforms. If the consistency testing is to be performed at the "implementation level," the mentioned constraint will contain parts that must be mapped to one platform and other parts that must be mapped to another platform.

As a full discussion of this type of mapping is beyond the scope of this paper, we simply assume that the legacy models in question are *implementation models* [12] and that the consistency testing is applied to instances of implementation models.

3 Modeling Consistency

In a consistency expression, <match> determines whether there is a relation between objects; and if so, <consistency test> decides the state of this relation: *consistent* or *not consistent*. All the presented techniques separate the determination of which objects to test and the actual consistency test. The matching is modeled with binary UML associations, which we term *consistency associations* (c-assoc). To separate consistency associations from other associations, the stereotype ⟪c-assoc⟫ is introduced—if allowed by the UML tool at hand, a dash-dotted line can be used as demonstrated in this paper. A class that is part of the consistency model is called a *consistency class* (c-class, stereotype ⟪c-class⟫). We have selected the following techniques as a starting point for our research:

- use of c-assoc between legacy classes (arbitrary multiplicity)
- use of c-assoc between legacy classes (arbitrary multiplicity), and an association class connected to the c-assoc
- use of c-classes that can be associated with several legacy classes through the use of c-assoc (arbitrary multiplicity on the legacy class end, only 0..1 or 1 on the c-class end)
- use of c-assoc between legacy classes (arbitrary multiplicity) and use of c-classes that can only be connected to one legacy class with an c-assoc (multiplicity is limited to 1 on the legacy class end, 0..1 or 1 on the c-class end)

For all techniques, the c-assoc and the constraint connected to the c-assoc (a missing constraint is the same as having a constraint that is always *true*) play

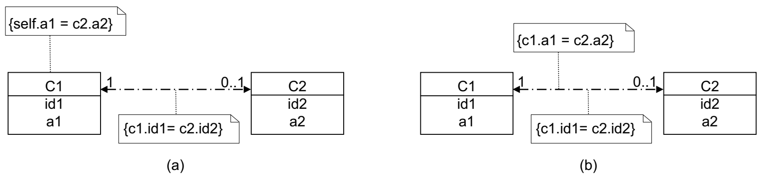


Fig. 3. Connection of Constraint to Class (a) and Association (b)

a key role when the consistency model is being instantiated. Logically, all possible instances of a c-assoc are considered when the consistency model is being instantiated. If the constraints on the c-assoc are met, the link is kept (if the c-assoc goes to a c-class, object of this class is created as needed). Note that this instantiation policy yields the consistency model with the maximum number of links. We have one important limitation on the constraints: *circular references between constraints must not occur*. Without this restriction several consistency model instances might be possible.

The different modeling techniques we consider will be tested on two examples. The first was given in Sect. 2.2. The second concerns a situation were one legacy system contains descriptions of apartments and another contains descriptions of buildings—see Fig. 4. The size of the floor space of a building should be the same as the total floor space of its apartments; the number of apartments that is given as an attribute in class Building should be equal to the number of apartments with the same building id (attribute bId). One building should have at least one apartment, and an apartment should belong to exactly one building. This simple example is sufficiently illustrative for our purposes. The following sections gives more details on the selected techniques.

3.1 Only Association and Constraints

The UML standard [2] states: “A constraint is a semantic condition or restriction expressed in text. In the metamodel, a Constraint is a BooleanExpression on an associated ModelElement(s), which must be true for the model to be well formed.” It is thus correct to attach a constraint to an association; OCL is not mentioned, and later on, OCL-invariants are only mentioned for classes, types, and stereotypes. But we assume that it is legal to connect a constraint to an association.

Fig. 3(b) shows an example where it is not possible to separate the matching from the actual consistency test. This observation reveals that this technique is not suited. In Fig. 3(a), the matching is connected to the c-assoc, and the consistency test is connected to one of the classes. The multiplicity is also of significance—in this example, an object of type C2 must be attached to an object of type C1 (with matching id); otherwise, there is an inconsistency. The next case is shown in Fig. 4.

As already mentioned the ”matching part” is not just an invariant, it is also a production rule. When consistency is to be tested, the consistency associations

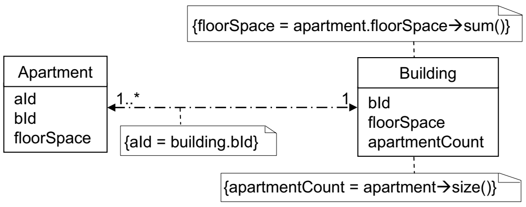


Fig. 4. Building with Apartments

will be instantiated, the instantiation policy will be to create all links that satisfy the match. If the specified multiplicity is broken (wrong number of links), it represents a consistency violation, which will be reported. Ideally, the matching should discriminate all links that are logically *wrong*, and it should include all the *right* ones. This would be the perfect match. A matching that is not discriminating enough might not be a problem: the links might not be used, or consistency tests that use them might always evaluate to true.

3.2 Consistency Modeled with Association Class

This technique extends the technique demonstrated in Fig. 3(b). An association class has been introduced to describe the consistency; Fig. 5 gives an example.

The constraint placed on the association corresponds to the matching.

The consistency test is formulated as an OCL expressed invariant on the association class; the attribute isConsistent, termed a consistency attribute, must be true if the considered objects are to be regarded as consistent. In a more complex situation, the consistency check can be structured into several invariants, distributed over several consistency attributes. A consistency attribute can be used in the reporting process, and it can also be referenced in other constraints.

Fig. 6 offers a solution for the second example. One weakness of this solution is that the consistency attributes will be calculated once for each apartment of a building when once for each building would suffice.

3.3 Ordinary Class as Consistency Class

In Fig. 7, a c-class connects legacy classes. This technique has many similarities with the one presented by Friis-Christensen and Jensen [5], where the aim is

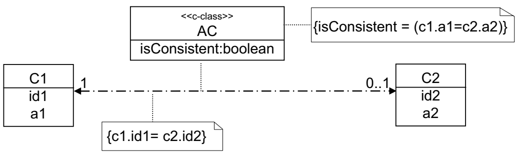


Fig. 5. Constraints Connected to An Association Class

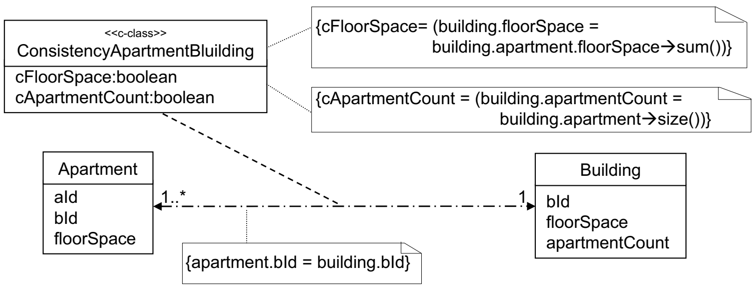

<<c-class>>
ConsistencyApartmentBluilding
cFloorSpace:boolean
cApartmentCount:boolean
{cFloorSpace= (building.floorSpace = building.apartment.floorSpace→sum())}
{cApartmentCount = (building.apartmentCount = building.apartment→size())}
Apartment
aId
bId
floorSpace
1..*
1
Building
bId
floorSpace
apartmentCount
{apartment.bId = building.bId}

Fig. 6. The Apartment / Building Problem Solved with An Association Class

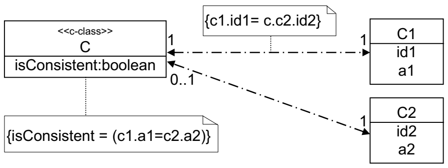


Fig. 7. Use of An Ordinary Class Instead of An Association Class

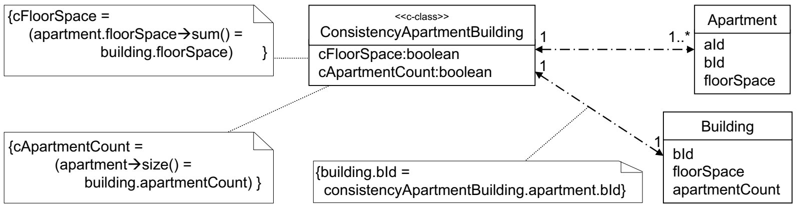

{cFloorSpace = (apartment.floorSpace→sum() = building.floorSpace) }
<<c-class>>
ConsistencyApartmentBuilding
cFloorSpace:boolean
cApartmentCount:boolean
1
1..*
1
1
Apartment
aId
bId
floorSpace
Building
bId
floorSpace
apartmentCount
{cApartmentCount = (apartment→size() = building.apartmentCount) }
{building.bId = consistencyApartmentBuilding.apartment.bId}

Fig. 8. Use of Ordinary Class: The Building / Apartment Example

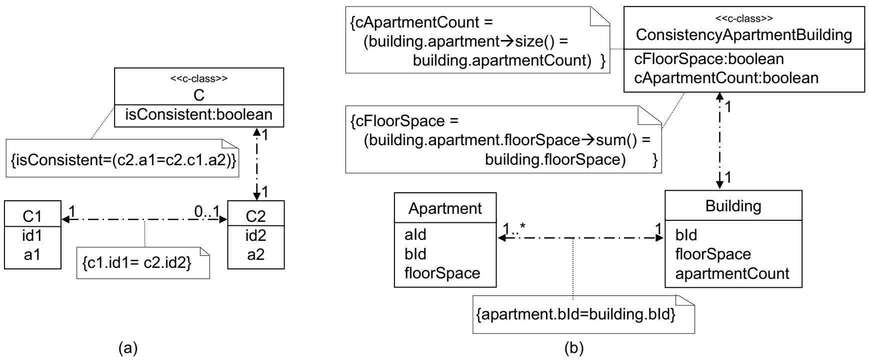
to integrate multiple representations of the same entity. But there are also differences. Whereas they place matching rules in a separate compartment of the c-class, we express the matching as constraints on the c-assoc.

Fig. 8 concerns the second example. The multiplicity on the apartment side in Fig. 8 demonstrates that several apartments are involved; the weakness found when association classes were used has been eliminated.

Yet another technique is demonstrated in Fig. 9. For each c-class, there is now exactly one c-assoc to a legacy class, and there can also be c-assoc's between legacy classes. It is also possible to put a constraint on the c-assoc from c-class to legacy class. This modeling technique seems to be simple and compact.

4 Selected Solution

Next, we consider which modeling technique to choose and how to consistency check user data.



<<c-class>>
C
isConsistent:boolean
{isConsistent=(c2.a1=c2.c1.a2)}
1
1
C1
id1
a1
1
0..1
C2
id2
a2
{c1.id1= c2.id2}
(a)
{cApartmentCount = (building.apartment→size() = building.apartmentCount) }
<<c-class>>
ConsistencyApartmentBuilding
cFloorSpace:boolean
cApartmentCount:boolean
{cFloorSpace = (building.apartment.floorSpace→sum() = building.floorSpace) }
1
1
Apartment
aId
bId
floorSpace
1..*
1
Building
bId
floorSpace
apartmentCount
{apartment.bId=building.bId}
(b)

Fig. 9. Consistency Classes with Only One Association to Legacy Class

4.1 Which Technique to Choose

Having introduced several modeling techniques, we proceed to compare these in order to understand their relative merits. A more exhaustive study, including a study of combinations of the techniques, is left for future research. The following evaluation criteria are important.

1. How easy is it to apply the technique and comprehend the resulting models?
2. Is it possible to make an interpreter (or compiler) for the models produced?
3. Is it possible to interpret the models efficiently?
4. Can good reports be made?

Criterion 1. We feel that all techniques are simple and rather intuitive; much of the required skill comes down to understanding OCL. But it seems that inserting c-assoc's directly between legacy classes is quite *natural.* It is not possible to avoid complex dependencies. Rather, the best one can do is to express them so that they are easily understood. The use of consistency attributes and c-assoc's makes it possible to partition complex OCL constraints into manageable pieces.

Criterion 2. Fulfillment of this point rests upon the possibility of instantiating the consistency model. This instantiation is later demonstrates for one technique; the other modeling techniques do not introduce any new complexity that cannot be solved by some extra mechanism for keeping track of intermediate results.

Criterion 3. A complex consistency model together with large amounts of user data can result in a combinatorial explosion, making the testing impossible in practice. Even if the presented techniques introduce only a limited number of new modeling elements, there will be practical limitations when it comes to the amount of user data to process. The technique that uses association classes seems to produce models that lead to much redundant testing. The last modeling technique allows the c-assoc links to be reused.

Criterion 4. Reporting is not elaborated upon in this paper. However, we note that the consistency attributes might play an important role in the reporting process.

The last modeling technique presented seems to be the best, and it is our choice. But we see the need for gaining more experience with the techniques, and a longer-term goal is to establish a framework that allows us to investigate the merits of the techniques.

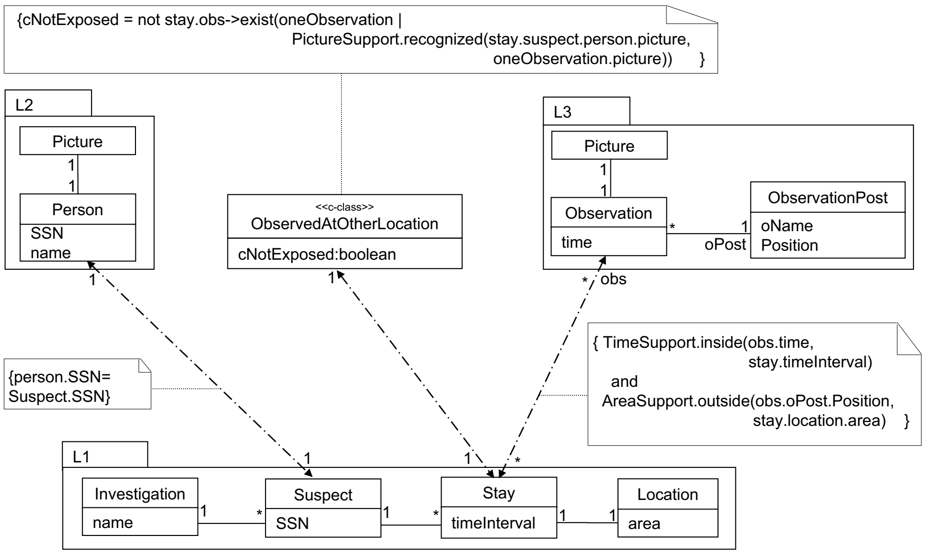


Fig. 10. Has the Suspect Been Elsewhere?

A more complex example is given in Fig. 10. Here, legacy system L2 contains pictures of persons. Legacy system L3 records observations done at different observation posts. Legacy system L1 contains information about police investigations. Instantiation and checking of the consistency model will expose a person who claims to have been one place, but has been observed in another place at the same time.

Classes TimeSupport, AreaSupport, and PictureSupport are part of the test environment; the operations of these classes that are used are all class scoped.

4.2 Consistency Testing Using the Chosen Technique

Fig. 11 presents a metamodel that illustrates which modeling elements to use and the constraints on them. In practice, the modeler must conform to the metamodel if the test is to be executed correctly. The metamodel also gives meaning to the stereotypes c-assoc and c-class.

We note that the c-assoc constraint is not shown in the metamodel, but it is important and will be described shortly. We proceed to explore the two main uses of the c-assoc:

1. It associates two legacy classes.
2. It associates a c-class and a legacy class.

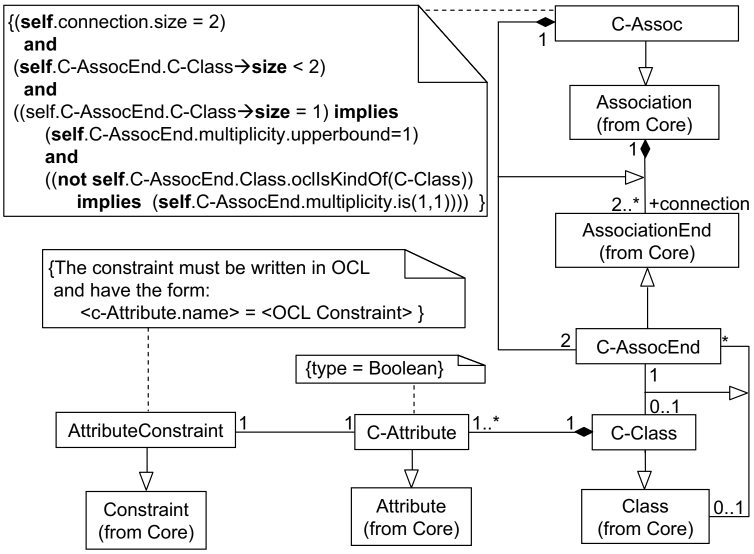

{(self.connection.size = 2)
and
(self.C-AssocEnd.C-Class→size < 2)
and
((self.C-AssocEnd.C-Class→size = 1) implies
(self.C-AssocEnd.multiplicity.upperbound=1)
and
((not self.C-AssocEnd.Class.oclIsKindOf(C-Class))
implies (self.C-AssocEnd.multiplicity.is(1,1)))) }
{The constraint must be written in OCL
and have the form:
<c-Attribute.name> = <OCL Constraint> }
{type = Boolean}
C-Assoc
1
Association
(from Core)
1
2..*
+connection
AssociationEnd
(from Core)
2
C-AssocEnd
*
1
0..1
AttributeConstraint
1
1
C-Attribute
1..*
1
C-Class
Constraint
(from Core)
Attribute
(from Core)
Class
(from Core)
0..1

Fig. 11. Metamodel

Use of c-assoc's with Legacy Classes. Legacy classes may belong to the same or different legacy systems. The c-assoc is similar to an ordinary association. Similarities include that role names can be inserted and that multiplicities can be selected freely. The only difference compared with an ordinary association is the c-assoc constraint: this constraint is a bit "more" than an ordinary association constraint. Like an ordinary constraint, it must be true for all links instantiated from the corresponding c-assoc; but, in addition, the c-assoc constraint is essential when instantiation of the c-assoc takes place. All possible links between objects of the two legacy classes are considered: the links that do not meet the c-assoc constraint are rejected; the rest are kept. If the number of links for an object do not meet the specified multiplicity, the cardinality is erroneous, and a consistency violation is reported. If a c-assoc constraint is absent, it will be the same as a c-assoc that is always true.

Use of c-assoc's with Legacy and c-classes. When a c-assoc associates a c-class and a legacy class, the multiplicity is "1" on the legacy class side and "1" or "0..1" on the c-class side.

The consistency checks are connected to a c-class through the consistency attributes. There can be many consistency attributes for each c-class. The value of a consistency attribute is given by the corresponding attribute constraint, which is of the form:

<c-Attribute.name> = <OCL constraint>

The <c-Attribute.name> is the name of the corresponding consistency attribute. The attribute constraint will always be true (as an invariant should). In addition, it is used when the attribute obtains its value. The value of the <OCL constraint> will be calculated, true or false, and the consistency attribute is given this value.

The association constraint connected to this c-assoc plays a role when instantiating the c-assoc and the c-class. Given an object of the legacy class, an object of the c-class will be created if the association constraint is met. If the multiplicity is "1" on the c-class side and no c-class object can be created because the constraint on the c-assoc cannot be meet, the cardinality is wrong, and a consistency violation is reported.

Cyclic References. A c-assoc may be referenced in attribute constraints and other c-assoc constraints. A constraint attribute may be referenced in attribute constraints that are attached to other c-classes (this is actually slightly too strict) and in c-assoc constraints. However, there is one limitation: the mentioned references must not be circular.

Consistency Model Instantiation. While most of the logic has already been described, one question remains, namely that of instantiation order. Since there can be references between the elements of the consistency model, this order cannot be arbitrary. The order can be decided by building a dependency graph (which will typically permit several orders). The nodes of the graph are obtained as follows.

1. A c-assoc that associates two legacy classes together with its constraint defines one node (node type 1).
2. An attribute constraint together with the c-assoc that associates its c-class and a legacy class defines a node (node type 2). If there are several attribute constraint for a class, there will be several nodes concerning the same class.

The edges arise from the navigations through c-assoc's and the references to the consistency attributes. If a node has a navigation that uses the c-assoc of another node, then there will be an edge from the first node to the second. Also, if one node has references to a constraint attribute (node of type 2) then there will be an edge from the first node to the second.

Since there are no cyclic references, the graph will be an acyclic directed graph. The instantiation can be done by selecting a node that fulfills the following: it has no edges pointing to nodes that have not already been instantiated. All instances of the selected node are created (e.g., if the node represents a c-assoc, all links with fulfilled constraints will be created).

The whole consistency model has been instantiated when there are no more nodes to instantiate. Use of the dependency graph ensures that all elements referenced in a constraint are present at instantiation time.

Instantiation of the first kind of nodes has already been explained. When instantiation of a node of type 2 is carried out, all legacy objects of the "right" kind are considered. The instantiation will occur in two ways:

– The c-class and corresponding c-assoc have not yet been instantiated. If the c-assoc constraint is met, an object of the c-class is created and linked to the legacy object by an instance of the c-assoc. The value of the consistency attribute is calculated, and the attribute is given this value. At this point, an object of the c-class has been created and linked to the legacy object; one consistency attribute has obtained its value.
– The c-class and corresponding c-assoc have been instantiated. The value of the consistency attribute is calculated, and the attribute is given this value.

4.3 Interpretation of OCL Expressions

The checking and evaluation of OCL expressions is done by an OCL interpreter. Building an interpreter or adapting an existing one for our purposes is achievable, as the object structure is stable and *ordinary*. An example of a rather similar application is found in the UML Specification Environment (USE) [13], where expressions written in OCL are used to specify integrity constraints on class diagrams. A model can be animated to validate the specifications; snapshots can be taken, and for each snapshot, the OCL constraints are checked automatically. Other tools for OCL includes the Dresden OCL Toolkit [14] and OCLE [15].

5 Summary and Research Directions

This paper has demonstrated how the full power of OCL as a declarative language can come to play in a setting where the consistency of semantically overlapping data sources is to be specified and checked. The proposed modeling technique is based on standard OCL and a small subset of UML's visual modeling elements. It is possible to use an ordinary UML tool to model the consistency. A consistency test environment is described. The use of XMI enables the integration of models and data of quite different origins; for this to happen, conversion to XMI must take place. The framework described needs a set of XMI-conversion tools to be in place; relation database schemas can easily be mapped to UML models, and it does not seem to be difficult to convert data stored in relational databases to XMI.

Because the amount of XML-data is growing rapidly, an investigation of how XML schemas and data fit into this framework is of great interest. The roles of ontologies [16] and the semantic web [17] have yet to be investigated; automatic generation of consistency models might be an option.

References

1. OMG Editor. *XML Metadata Interchange (XMI) Specification v1.2.* OMG Document. OMG,
http://www.omg.org, January 2002.
2. OMG Editor. *OMG Unified Modeling Language Specification, Version 1.5.* OMG Document. OMG,
http://www.omg.org, March 2003.
3. R Barták. Constraint Programming: In Pursuit of the Holy Grail. In *In Proceedings of Workshop on Distributed Systems, Prague, Czech Republic*, pages 555–564 1999.
4. S. Ceri and J. Widom. Managing Semantic Heterogeneity with Production Rules and Persistent Queries. In *International Conference on Very Large Data Bases*, pages 108–119, 1993.
5. A. Friis-Christensen and C. S. Jensen. Object-Relational Management of Multiply Represented Geographic Entitites. In *International Conference on Scientific and Statistical Database Management*, 2003, to appear.
6. M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying Interdatabase Dependencies in a Multidatabase Environment. *Computer*, 24(12): 46–53, 1991.
7. M. Casanova, T. Wallet, and M. D'Hondt. Ensuring Quality of Geographic Data with UML and OCL. LNCS 1939, pages 225–239, Springer, 2000.
8. OMG Editor. *Model Driven Architecture.* OMG document number ormsc/2001-07-01. (July 2001) OMG,
http://www.omg.org/docs/ormsc/01-07-01.pdf.
9. A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2002.
10. K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1): 63–75, 1985.
11. S. Gaito, S. Kent, and N. Ross. A Meta-Model Semantics for Structural Constraints in UML. *Behavioural Specifications for Businesses and Systems*, pages 123–141, September 1999.
12. M. Fowler and S. Kendall. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley, 2nd edition, 2002.
13. M. Gogolla and M. Richters. Development of UML Descriptions with USE. In *First Eurasian Conference on Information and Communication Technology*, LNCS 2510, pages 228–238. Springer, 2002.
14. H. Hussmann, B. Demuth, and F. Finger. Modular Architecture for a Toolset Supporting OCL. In *<<UML>>2000*, pages 278–293, Springer, 2000.
15. Computer Science Research Laboratory of Babes-Bolyai University of Cluj-Napoca Romania.
http://lci.cs.ubbcluj.ro/ocle/, April 2003
16. K. Baclawski, M. K. Kokar, P. A. Kogut, L. Hart, J. Smith, W. S. Holmes III, J. Letkowski, and M. L. Aronson. Extending UML to Support Ontology Engineering for the Semantic Web. LNCS 2185, pages 342–360, Springer, 2001.
17. Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.