Verification Technology for Object-Oriented/XML Transactions

Suad Alagić, Mark Royer, and David Briggs

Department of Computer Science
University of Southern Maine
Portland, ME 04104-9300
{alagic,mroyer,briggs}@usm.maine.edu

Abstract. Typically, object-oriented schemas are lacking declarative specification of the schema integrity constraints. Object-oriented transactions are also typically missing a fundamental ACID requirement: consistency. We present a developed technology based on object-oriented assertion languages that overcomes these limitations of persistent and database object systems. This technology allows specification of object-oriented integrity constraints, their static verification and dynamic enforcement. Proof strategies that are based on static and dynamic verification techniques as they apply to verification of object-oriented transactions are presented in the paper. Most of this work has been motivated by the problems of object-oriented interfaces to XML that have not been able to express typical XML Schema constraints, database constraints in particular. The components of this technology are an object-oriented constraint language, a verification system with advanced typing and logic capabilities, predefined libraries of object-oriented specification and verification theories, and an extended virtual platform for integrating constraints into the run-time type system and their management.

1 Introduction

Most persistent object and object database technologies lack the ability to express the schema integrity constraints in a declarative fashion, as is customary in conventional data models. The reason is that the mainstream object-oriented languages lack such declarative logic-based specification features. Specification of even the most typical database constraints is beyond expressiveness of object-oriented type systems of mainstream object-oriented languages. This is why the notion of a transaction in most object-oriented technologies does not include a fundamental ACID requirement: consistency. Since object-oriented schemas do not contain specification of general integrity constraints (and often not even keys and referential integrity) requiring that a transaction should satisfy those constraints becomes very problematic.

These limitations emerge when interfacing object-oriented technology with XML. Virtually all object-oriented interfaces to XML, as well as typed XML oriented languages, suffer from the inability to express constraints such as those

M.C. Norrie and M. Grossniklaus (Eds.): ICOODB 2009, LNCS 5936, pp. 23–40, 2010.
© Springer-Verlag Berlin Heidelberg 2010

available in XML Schema [22]. These constraints include specification of the ranges of the number of occurrences, keys and referential integrity. A core idea behind type derivations in XML Schema is that an instance of a derived type may be viewed as a valid instance of its base type. This includes the requirement that all constraints associated with the base type are still valid when applied to an instance of a derived type.

Overcoming these limitations becomes possible with the proliferation of object-oriented assertion languages, such as JML [14] or Spec# [7]. Object-oriented assertion languages now allow both specification of the integrity constraints in object-oriented schemas and enforcing them when executing database transactions. In addition to specifying constraints that are sufficient for XML types, JML allows specification of mutation (update) of database state. Moreover, the notion of a transaction that updates the database state maintaining the integrity constraints of the database schema can now be specified in this technology. In fact, if the actual Java code is provided, it will be possible to enforce the requirement that a transaction must comply with the integrity constraints.

The availability of constraints makes it possible to use a prover technology for automated reasoning about a variety of properties expressed by constraints. This applies even to application properties that are not expressible in XML Schema. Thus, reasoning and verification are supported in situations when XML data is processed by a transaction or a general purpose programming language. While dynamic enforcement of constraints is a reality in the actual systems, our goal is to use a suitable prover technology to carry out deductions to statically verify properties expressed by constraints.

Our choice of PVS (Prototype Verification System) [15] is based on its sophisticated type system (including subtyping and bounded parametric polymorphism) accommodating a variety of logics with higher-order features. A PVS specification consists of a collection of theories. A theory is a specification of the required type signatures (of functions in particular) along with a collection of constraints in a suitable logic applicable to instances of the theory. Since PVS is a higher-order system it allows embedding of specialized logics as we did for temporal logic, applying the result to Java classes [2].

Our proof methodology for verification that a transaction respects the integrity of a schema equipped with constraintsrequires explicit specification of the frame constraints of a transaction. The frame constraints specify the integrity constraints which the transaction does not affect. In addition, the active part (the actual update) that a transaction performs is specified in a declarative, logic-based style, and the verification is carried out using a proof strategy presented in the paper. This methodology is independent of a particular transaction language [3]. Previous work on transaction verification includes [18,19,8,9,5].

The paper is organized as follows. In Sect. 2 we introduce a motivating example which illustrates the main problems in object-oriented representation of XML Schema constraints. Section 3 presents on overview of the architecture of our underlying software technology for specification, representation and management of constraints and their static and dynamic enforcement. Sections 4, 5, 6, and 7

show how XML Schema constraints and transactions are specified using JML. In Sect. 8 and 9 we present our techniques for representing JML specifications in PVS. This is followed by the transaction verification techniques presented in Sect. 10. Section 11 shows how our extended virtual platform contributes to the overall technology for management of constraints, and their enforcement in transaction verification.

2 Motivation: XML Schema Constraints

Although the technology presented in this paper is a general object-oriented constraint technology, a substantial part of the motivation comes from the problems of interfacing object-oriented persistent and database technology with XML Schema [13]. A typical XML Schema constraint specifies the range of the number of occurrences of an XML term (an element or a group). This type of a constraint is illustrated below by a type `XMLproject` specified according to the XML Schema formalism.

```
<xsd:complexType name = "XMLproject"
    <xsd:sequence>
     <xsd:element name = "leader" type = "XMLprojectLeader" />
     <xsd:element name = "funds"  type = "xsd:positiveInteger" />
     <xsd:element name = "contract" type = "XMLcontract"
                            minOccurs = "1" maxOccurs = "5" />
    </xsd:sequence>
    <xsd:attribute name = "projectId" type = "xsd:string" />
</xsd: complexType>
```

XML Schema comes with two techniques for type derivation: by extension and by restriction. Type derivation by extension can be represented using inheritance in spite of some subtleties. However, object-oriented interfaces to XML cannot represent type derivation by restriction because this form is, among other subtleties, based on restricting the range constraints of the base type in the type derived by restriction. This is illustrated below by a type `XMLspecialProject` derived by restriction from the type `XMLproject`. A special project is required to have exactly one contract.

```
<xsd:complexType name = "XMLspecialProject"
   <xsd:complexContent>
    <xsd:restriction base = "XMLproject" >
     <xsd:sequence>
       <xsd:element name = "leader" type = "XMLprojectLeader" />
       <xsd:element name = "funds"
                    type = "xsd:positiveInteger" />
       <xsd:element name = "contract" type = "XMLcontract"
                              minOccurs = "1" maxOccurs = "1" />
     </xsd:sequence>
   <xsd:attribute name = "projectId" type = "xsd:string" />
   </xsd:complexContent>
</xsd:complexType>
```

A sample application schema in this paper consists of a sequence of projects and a sequence of contracts. Specification of these two types in the XML Schema formalism is given below. The range-of-occurrences constraints are such that representing these types in object-oriented interfaces would not be a problem using parametric types such as a sequence or a list. But if the range constraints were more specific like in `XMLproject` and `XMLspecialProject`, object-oriented interfaces could not represent them.

```
<xsd:complexType name = "XMLsequenceOfProjects"
     <xsd:sequence>
      <xsd:element name = "project" type = "XMLproject"
                   minOccurs = "0" maxOccurs = "unbounded" />
     </xsd:sequence>
</xsd:complexType>
<xsd:complexType name = "XMLsequenceOfContracts"
     <xsd:sequence>
      <xsd:element name = "contract" type = "XMLcontract"
                   minOccurs = "0" maxOccurs = "unbounded" />
     </xsd:sequence>
<xsd:complexType>
```

XML Schema also allows specification of typical database integrity constraints such as keys and referential integrity. Project and contract keys are specified below according to the XML Schema formalism, so that the attribute `projectId` is a key for the sequence of projects and `contractNo` is a key for the sequence of contracts. Object-oriented interfaces to XML such as DOM [11], LINQ to XML [21] and LINQ to XSD [20] are constrained by the limitations of object-oriented type systems. This is why they have no way of specifying any of these constraints, because these constraints are not expressible in the standard object-oriented type systems. The same applies to referential integrity constraints in XML Schema illustrated below. This referential constraint specifies that the contract numbers of contracts of a project must be valid, i.e., keys that actually appear in the sequence of contracts.

```
<xsd:element name= "allContractsAndProjects">
    <xsd:complexType>
       <xsd:sequence>
          <xsd:element name = "Contracts"
                       type ="XMLSequenceOfContracts" />
          <xsd:element name = "Projects"
                       type = "XMLSequenceOfProjects" />
       </xsd:sequence>
    </xsd:complexType>

    <xsd:key name ="contractKey" />
       <xsd:selector xpath="./Contracts/contract" />
       <xsd:field xpath="@contractNo" />
    </xsd:key>
    <xsd:key name ="projectKey" />
```

```
      <xsd:selector xpath="./Projects/project" />
      <xsd:field xpath= "@projectId" />
   </xsd:key>
   <xsd:keyref name = "projectToContract" refer="contractKey">
      <xsd:selector xpath ="Projects/project/contract" />
      <xsd:field xpath ="@contractNo" />
   </xsd:keyref>
</xsd:element>
```

The new proposal for XML Schema 1.1 [22] includes even more general constraints specified as assertions that are based on Xpath expressions. Complex applications naturally contain other types of constraints that cannot be expressed in the XML Schema formalism and well-known object-oriented interfaces to XML cannot represent them either. The problem here is that object-oriented interfaces to XML are used in complex object-oriented software application packages that should enforce the application constraints.

3 Architecture

The underlying support of this technology is an extended virtual platform (XVP) implemented in a related project [17]. This platform allows declarative representation of constraints, introspection by extended reflective capabilities that report constraints along with the type signatures, and interfacing with a program

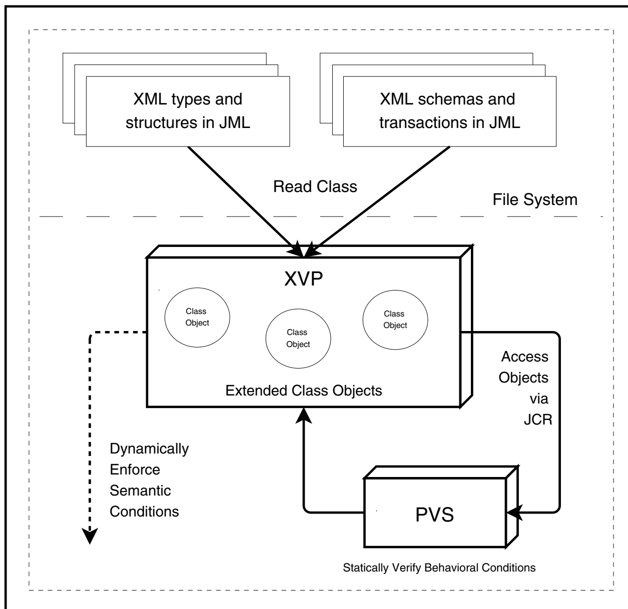

Fig. 1. Components of the technology

verification system. The components of this technology represented in Fig. 1 inter-operate as follows.

– Application schemas are specified in JML by extending our predefined library of JML specifications of the core of XML Schema.
– Application programs and transactions are also specified in JML and implemented in Java, so that JML constraints will be enforced at run-time.
– JML specifications are compiled by a special compiler [17]. The extended virtual platform [17] makes constraints available for introspection and enforcement.
– In order to carry out static verification the PVS theories relevant for the verification task are produced extending our predefined library of PVS theories for the XML Schema core.

This architecture makes a variety of verification techniques possible by combining static and dynamic techniques. If the constraints to be enforced dynamically are taken as assumptions, other constraints, database integrity in particular, may be provable statically. If so, the latter constraints will not have to be verified at run-time, increasing efficiency and reliability of transactions.

4 Object-Oriented Assertions

The JML representation of an XML element `XMLproject` is specified below. The three components of this specification are: the type information associated with an element type, the type signatures of accessor functions, and the constraints. In fact, the availability of constraints makes this representation possible. A complete and correct representation would not be possible in a type system alone as it requires constraints, like those expressible in JML.

A project element consists of three subelements (`leader`, `contract` and `funds`), and a single attribute (`projectId`). These are specified in the inner types (classes) `ProjectElements` and `ProjectAttributes`. The function `elements` returns project elements and the function `attributes` returns project attributes. Usage of parametric types appears in comments because their support in JML is still under development.

```
//   class XMLproject extends XMLelement<XMLcomplex>
public class XMLproject extends XMLelement {
    /*@ ensures this.elements().fundsConstraint() &&
                 this.elements().rangeConstraint(); @*/
XMLproject(ProjectElements elements,
                      ProjectAttributes attributes) {. . . }
/*@ pure @*/
public ProjectElements elements() { . . . }
/*@ pure @*/
public ProjectAttributes attributes() {. . . }

/*@ ensures \result <==> ((XMLfloat)this.elements().funds().
                     value()).floatValue() >= 100000;  pure  @*/
```

```
public boolean fundsConstraint() {. . . }

/*@ ensures \result <==> this.elements().contract().occurs() >= 1
             && this.elements().contract().occurs() <= 5; pure @*/
public boolean rangeConstraint() {. . . }

/* specification of ProjectElements and ProjectAttributes */

/*@ invariant this.fundsConstraint() && this.rangeConstraint(); @*/
}
```

The JML representation technique of type derivation by restriction as defined in XML Schema is illustrated below. The type `XMLspecialProject` extends the type `XMLproject` using inheritance as specified in Java. There are no new components of `XMLspecialProject` in comparison with `XMLproject`, but the constraints in `XMLspecialProject` are strengthened with respect to the constraints in `XMLproject`. This corresponds to the XML Schema notion of type derivation by restriction, except that the constraints in our JML and PVS based technology can be much more general.

```
public class XMLspecialProject extends XMLproject {
// . . .
 /*@ also ensures this.fundsConstraint() <==>
 ((XMLfloat)this.elements().funds().
                value()).floatValue() >= 1000000;  pure @*/
 public boolean fundsConstraint() {. . .}

/*@ invariant this.elements().contract().maxOccurs() == 1; @*/
}
```

5 Application Schemas

A project management application schema `XMLprojectManagement` contains a sequence of contracts and a sequence of projects. This specification contains two constraints typical for database schemas and available in XML Schema. The `uniqueness` constraint specifies that contract numbers uniquely determine contracts in the sequence of contracts. The `referential` constraint specifies that contracts of projects in the sequence of projects exist in the sequence of contracts. In addition to the above two XML Schema types of constraints, the `ordering` constraint specifies that contracts appear in the sequence of contracts in increasing order of their contract numbers. There is also a self-explanatory `fundsRange` constraint. The `ordering` and the `fundsRange` constraints are samples of typical database constraints. But the advantage of using a general constraint language such as JML is that we can express more general constraints belonging to the application environment and enforce them. Application requirements typically go beyond the expressive capabilities of the constraint language for XML Schema or conventional database management systems.

```
public class XMLprojectManagement implements XMLschema {
/*@ pure @*/
public XMLsequence projects() {. . . }
     // XMLsequence<XMLproject> projects();
/*@ pure @*/
public XMLsequence contracts() {. . . }
     // XMLsequence<XMLcontract> contracts();

/*@ ensures \result <==> (\forall XMLcontract c1,c2;
              contracts().member(c1) && contracts().member(c2) &&
  c1.attributes().contractNo().equals
        (c2.attributes().contractNo()) ==> c1.equals(c2)); pure @*/
public boolean uniquenessConstraint() { . . .}

/*@ ensures \result <==> (\forall XMLproject p;(\forall XMLcontract c;
    projects().member(p) && p.elements(). contract().equals(c) ==>
               (\exists XMLcontract c1; contracts().member(c1) &&
                        c.attributes().contractNo().equals
                       (c1.attributes().contractNo())))); pure @*/
public boolean referentialConstraint() { . . . }

/*@ ensures \result <==>(\forall XMLcontract c1,c2;(\forall int n1,n2;
                 contracts().member(c1) & contracts().member(c2) &&
    c1.attributes().contractNo() <= c2.attributes().contractNo() &&
                                  contracts().get(n1).equals(c1) &&
              contracts().get(n2).equals(c2) ==> n1 <= n2)); pure @*/
public boolean orderingConstraint() { . . . }

/*@ ensures \result <==> (\forall XMLproject p;
      projects().member(p) ==> p.fundsConstraint()); pure @*/

public boolean fundsRangeConstraint() {. . .}

/*@ also ensures \result <==> this.uniquenessConstraint() &&
         this.referentialConstraint() && this.orderingConstraint() &&
         this.fundsRangeConstraint();         pure @*/
public boolean consistent() { . . . }

/*@ invariant this.consistent(); @*/
}
```

6 Data Manipulation via Mutator Methods

Object-oriented interfaces to XML are largely intended for developing applications that manipulate the object-oriented representation of XML data. This is where the availability of constraints is critical to maintain data integrity. The existing object-oriented interfaces such as DOM [11], LINQ to XML [21] and LINQ to XSD [20] have no way of enforcing constraints of XML Schema in data

manipulation actions. This becomes possible using object-oriented assertion languages. A few illustrative examples follow.

```
/*@ ensures this.fundsRangeConstraint(); @*/
void updateFunds(XMLelement amount) {. . .}
              // XMLelement<Float> amount

/*@ ensures this.fundsRangeConstraint() &&
            this.referentialConstraint(); @*/
void updateProjects(XMLsequence projects){. . .}
                 // XMLsequence<XMLproject>

/*@ ensures this.uniquenessConstraint() &&
            this.orderingConstraint(); @*/
void updateContracts(XMLsequence contracts) {. . . }
                  // XMLsequence<XML contract>
```

The above examples demonstrate some of the major advantages of the constraint-based approach with respect to the previous results. Enforcing the integrity constraints is a critical issue for database transactions that perform data manipulation. This cannot be accomplished with other approaches that are not based on constraints, but rather on type systems alone.

7 Transactions

Our JML specification of the class `Transaction` shares some similarity with the ODMG specification, but the ODMG specification does not have two critical ingredients: constraints and bounded parametric polymorphism [10]. The type constraint says that the actual type parameter must extend the type `XMLschema`. This is how a transaction is bound to its schema. In spite of all problems related to genericity in Java [1], this form of bounded parametric polymorphism is supported in the recent editions of Java. The JML assertions make it possible to specify the requirements that a transaction must respect the schema integrity constraints.

```
// abstract class XMLtransaction <T extends XMLschema>
abstract public class XMLtransaction {
// . . .
 /*@ pure @*/
 abstract XMLschema schema(); // T schema()
}
```

A specific transaction is specified below. The fact that this transaction is defined with respect to the `XMLprojectManagement` schema is represented using `XMLprojectManagement` as the actual type parameter. The constructor takes an instance of the `XMLprojectManagement` schema and makes it the schema of this transaction returned by the method `schema`.

The actual update that the transaction performs is specified in the method `update`. This method requires that the schema consistency requirements are satisfied before the update is executed. One of the conditions that this method ensures after its execution is that the sequences of contracts before and after execution of `update` are equal. In other words, this transaction does not affect the sequence of contracts. In addition, the method `update` ensures that the referential integrity constraint of the `XMLprojectManagement` schema holds after method execution. The remaining part of the postcondition specifies the actual update that the transaction performs which is increasing project funds by the specified amount.

```
// class XMLprojectTransaction
//                         extends XMLtransaction<XMLprojectManagement>
public class XMLprojectTransaction extends XMLtransaction {
 XMLprojectTransaction(XMLprojectManagement schema) {. . . }

 /*@ pure @*/
 XMLprojectManagement schema(){. . . }

 /*@ ensures this.schema().contracts().equals(
                                \old(this.schema().contracts()))
     && this.schema().referentialConstraint() &&
  (\forall int n; 1 <= 1 && n <= this.schema().projects().length();
  ((XMLfloat)((XMLproject)this.schema().projects().get(n)).elements().
                     funds().value()).floatValue() ==
  \old(((XMLfloat)((XMLproject)this.schema().projects().get(n)).
             elements().funds().value()).floatValue()) + 1000 );  @*/
 void update(float increase) {. . . }
 }
```

Note that the result type of the method `schema` has been overridden covariantly as in the recent editions of Java. If the method `update` is implemented in Java, JML will dynamically enforce the above requirements. In the above example an obvious question is whether the schema integrity constraints will indeed be satisfied if a transaction behaves according to the above specification. As even this simple example shows, when the integrity constraints and transaction updates become more complex, their verification requires support from a suitable prover technology.

8 PVS Theories

In order to use the PVS prover, JML specifications must be transformed into PVS theories, preferably by an automated tool. A PVS theory is a specification of the required type signatures (of functions in particular) along with a collection of constraints in a suitable logic applicable to instances of the theory. Our core techniques include representation of inheritance, method overriding and parametric types.

PVS does not support the object-oriented notion of inheritance. Our PVS representation technique for inheritance has the following form:

```
A: THEORY
BEGIN A: TYPE
% body of theory A
END A
```

```
B: THEORY
BEGIN  IMPORTING A
       B: TYPE FROM A
 % body of theory B
END B
```

In PVS the subtype declaration `B: TYPE FROM A` is equivalent to

```
B_pred: [A -> bool]
B: TYPE =(B_pred)
```

where `(B_pred)` denotes a type that satisfies `B_pred`. This is the PVS notion of predicate subtyping. The implications of the PVS notion of predicate subtyping on modeling inheritance of methods are elaborated in [2].

The PVS notion of predicate subtyping has the following implication on modeling inheritance of methods. A method `m` of `A` with the signature
`m: [A,C2,...,A,...,Cm -> A]`
will be available in `B` with exactly the same signature, just like in the Java invariant subtyping rule for signatures of inherited methods. However, since `B` is a PVS subtype of `A`, the effect would be as if `m` is available in `B` with the signature `m: [B,C2,...,B,...,Cm -> A]`. Otherwise, overriding the signature of `m` in `B` to a signature such as
`m: [B,C2,...,B,...,Cm -> B]`
which has covariant change of the result type as in recent editions of Java requires definition of a new function `m` in `B`.

Unlike the current version of JML, PVS supports parametric and even bounded parametric polymorphism. A theory representing a parametric type `C` with a bounded type constraint has the following form:
`C[(IMPORTING B) T: TYPE FROM B]`

The fact that a theory `K` with a bound `B` for its type parameter `T` is representing a subclass of a parametric class `C` is represented in the PVS notation as follows:

```
K [(IMPORTING B) T: TYPE FROM B]
BEGIN
      IMPORTING C[T]
      K: TYPE FROM C[T]
% body of K
END K
```

9 Application-Oriented PVS Theories

Application-oriented PVS theories are illustrated in the specification `XMLproject` given below. The type information for subelements and attributes is represented by record types. However, because of repetition of the subelement

`contract`, `XMLproject` is not represented as a record, since that would not be an accurate representation with respect to XML. `contract` is a unique identifier in the record structure, and it gets repeated as the tag of any occurence of this subelment in the `XMLproject` element. The repetition is expressed via `minOccurs` and `maxOccurs` constraints and also by specifying the tag language of `XMLproject`. In addition to the above two components (type structure and constraints), the third component consists of accessor functions that apply to an instance of an `XMLproject`. Note that the accessor function `projectContracts` returns a sequence of `XMLcontract` elements.

```
XMLproject: THEORY
BEGIN
     IMPORTING  XMLcomplex, XMLcontract, XMLstring
     XMLproject: TYPE+ FROM XMLelement

  XMLprojectElements: TYPE  =  [# leader: string, funds: real,
                                  contract:  XMLcontract #]
  XMLprojectAttributes: TYPE = [# projectId: string #]

  project: [XMLprojectElements, XMLprojectAttributes -> XMLproject]
  elements:   [XMLproject -> XMLprojectElements]
  attributes: [XMLproject -> XMLprojectAttributes]

 p: VAR XMLproject
 leader(p): string = leader(elements(p))
 funds(p): real    = funds(elements(p))
 contract(p): XMLcontract = contract(elements(p))
 projectContracts: [XMLproject -> XMLsequence[XMLcontract]]

 fundsConstraint(p: XMLproject): bool =
                              (funds(elements(p))) >= 1000000

 contractElementsConstraint(p: XMLproject): bool =
           minOccurs(contract(elements(p))) >= 1 AND
           maxOccurs(contract(elements(p))) = unbounded

 elementTags(p: XMLproject): XMLtags =
           conCat(singleton(seq("leader")),
               conCat(singleton(seq("funds")),

                      starPlus(singleton(seq("contract")))) )
END XMLproject
```

Specification of the `XMLprojectManagement` schema now follows the initial JML specification. The constraints specify the schema consistency requirements discussed earlier in the PVS notation.

```
XMLprojectManagement: THEORY
```

```
 BEGIN
 IMPORTING XMLcomplex, XMLcontract, XMLproject, XMLsequence, XMLschema
 XMLprojectManagement: TYPE+  FROM XMLschema

  projects:   [XMLprojectManagement -> XMLsequence[XMLproject]]
  contracts:  [XMLprojectManagement -> XMLsequence[XMLcontract]]

M: VAR XMLprojectManagement
p: VAR XMLproject
c: VAR XMLcontract

uniquenessConstraint(M): bool =  (FORALL (c1,c2: XMLcontract):
       member(contracts(M),c1) AND member(contracts(M),c2) AND
       contractNo(contractAttributes(c1)) =
           contractNo(contractAttributes(c2))  IMPLIES c1 = c2)

referentialConstraint(M): bool = (FORALL (p,c):
  (member(projects(M),p) AND contract(elements(p)) = c) IMPLIES
    (EXISTS (c1:XMLcontract):(member(contracts(M),c1) AND
             (contractNo(contractAttributes(c1)) =
                            contractNo(contractAttributes(c))))))

orderingConstraint(M): bool = (FORALL (c1,c2: XMLcontract,
                               n1,n2: below(length(contracts(M)))):
               member(contracts(M),c1) AND member(contracts(M),c2)
                  AND  contractNo(contractAttributes(c1)) <=
                            contractNo(contractAttributes(c2)) AND
                            nth(contracts(M))(n1) = c1 AND
                        nth(contracts(M))(n2) = c2 IMPLIES n1 <= n2)

 fundsRange(M): bool = (FORALL (n: below(length(projects(M)))):
                            fundsConstraint(nth(projects(M))(n)))

 consistent(M): bool = uniquenessConstraint(M) AND
                       referentialConstraint(M) AND
                       orderingConstraint(M)  AND
                       fundsRange(M)
END XMLprojectManagement
```

10 Transaction Verification in PVS

A transaction theory `XMLprojectTransaction` contains specification of both the frame constraint and the actual update that the transaction performs [3]. The frame constraint specifies the integrity constraints that are not affected by the transaction. This particular transaction only updates contract funds and hence it has no impact on the uniqueness, referential, and ordering constraints. Explicit specification of the frame constraints is essential in our proof strategy that guides the prover appropriately. The actual update that the transaction

performs is specified in a declarative fashion as a predicate over a pair of object states, the state before and the state after transaction execution. A transaction is then a binary predicate specified as a conjunction of its frame constraint and the actual update constraint.

```
XMLprojectTransaction: THEORY
BEGIN
     IMPORTING XMLtransaction, XMLprojectManagement
     XMLprojectTransaction: TYPE FROM XMLtransaction
     M1,M2: VAR XMLprojectManagement

 frameAx(M1,M2): bool = consistent(M1) AND
         contracts(M1) = contracts(M2) AND referentialConstraint(M2)

 update(M1,M2): bool =  length(projects(M1)) = length(projects(M2))
                      AND  FORALL (n: below(length(projects(M2)))):
                         (funds(elements(nth(projects(M2))(n))) =
                    funds(elements(nth(projects(M1))(n))) + 100000)

 transaction(M1,M2): bool = frameAx(M1,M2) AND update(M1,M2)
END XMLprojectTransaction
```

In order to prove that a transaction which conforms to the above theory maintains the integrity of the `XMLprojectManagment` database, the following theory is constructed. To simplify the proof, a simple update lemma is proved first. The integrity theorem is then proved using the update lemma [3].

```
VerifyProjectTransaction: THEORY
BEGIN
       IMPORTING XMLprojectTransaction
       M1,M2: VAR XMLprojectManagement

updateLemma: LEMMA fundsRange(M1) AND update(M1,M2)
                         IMPLIES  fundsRange(M2)
Integrity: THEOREM FORALL (M1,M2):
       consistent(M1) AND transaction(M1,M2) IMPLIES consistent(M2)
END VerifyProjectTransaction
```

Consider an example of a characterization of a transaction update that violates the referential integrity constraint and hence its `Integrity` theorem fails. Let us define `badUpdate` as

```
badUpdate(M1,M2): bool = length(projects(M1)) > 0 AND
       projects(M2) = projects(M1) AND length(contracts(M2)) = 0
```

This update does not affect the sequence of projects but it deletes all contracts which is an obvious violation of referential integrity. The PVS proof of the `updateLema` leads to an obvious contradiction demonstrating violation of integrity.

11 Virtual Platform Support

In this section we show an example of combination of static and dynamic verification of a transaction that relies on the support of the extended virtual platform [17]. The main components of an extended virtual platform are given in Fig. 2.

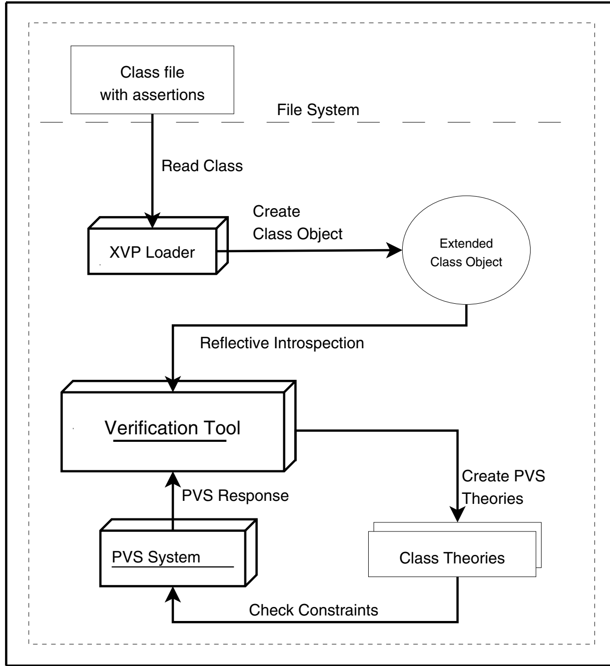

Fig. 2. Verification in the extended virtual platform

The existing Java reflective capabilities allow introspection of type signatures and the extended virtual platform allows introspection of the constraints associated with those types. Constraints are reported by the extended reflective capabilities in their logic-based declarative style. This is a major distinction in comparison with existing virtual platforms, and JML in particular.

This system is designed in such a way that it is independent of a particular constraint language and its underlying logic basis. The program verification system accesses loaded class objects through a tool that makes use of extended reflective capabilities. The interface component produces a program verification theory of a class and the program verification system carries out deduction and reports the results.

The Java Core Reflection (JCR) classes that have been extended are `Class`, `Constructor`, and `Method`. These extensions are based on new types such as

Invariant, PreCondition, and PostCondition. With these new types it becomes possible to add method preconditions and postconditions to the class Method, and the class invariant to the class Class. These assertions require further types that make it possible to create objects that represent logical formulas for constraints. In order to achieve independence of a particular constraint language and its logic basis, the types representing logical formulas are specified as abstract classes. These classes must be extended for a particular assertion language as we did for JML.

Additions of the recompiled class Class allow access to the declared and the inherited invariant. The extensions of the class Method allow access to (declared and inherited) preconditions and postconditions. The class Constructor is similarly extended.

```
public final class Class { ...
  public Invariant getInvariant();
}
public final class Method { ...
 public PostCondition  getPostCondition();
 public PreCondition   getPreCondition();
}
```

Each one of the above assertion types is also equipped with the method evaluate for run-time evaluation of the assertion objects. The differences in the signatures of these methods are in their parameters and reflect the nature of the assertions. Postcondition is also equipped with a method to bind parameters taking into account both the object state before and after method execution.

In the example given below, the method executeProjectTransaction is written with the assumption that static verification of the transaction proTransaction has been carried out. Static verification proves that if proManagement is consistent and the transaction proTransaction is executed, proManagement will remain consistent. The consistency predicate is in fact the invariant of the object proManagement.

Dynamic verification requires access to the precondition of the method update of proTransaction and the invariant of the object proManagement. This is done by extended reflection. If these two conditions evaluate to true, the update method is invoked using standard reflection. Upon execution of this method its postcondition is evaluated using extended reflection. If it evaluates to true then we know that the invariant will hold and hence it does not have to be checked dynamically. If the postcondition does not hold the transaction proTransaction is aborted. Otherwise it is committed.

```
executeProjectTransaction(XMLprojectManagement proManagement,
                          XMLprojectTransaction proTransaction,
                          XMLfloat amount) throws Exception
{ proTransaction.start();
  Invariant inv = proManagement.getClass().getInvariant();
  Method update = proTransaction.getClass().getMethod("update",
                                    new Class[]{XMLfloat.class});
```

```
  Object[] params = new Object[]{amount};
  PreCondition preCond = update.getPreCondition();
  PostCondition postCond = update.getPostCondition();
  postCond.bindPreMethodVars(proTransaction,params);

  if (inv.evaluate (proManagement) &&
              preCond.evaluate(proTransaction,params))
      update.invoke(proTransaction,params);

   if (postCond.evaluate(proTransaction,null,params))
      proTransaction.commit();
   else proTransaction.abort();
}
```

12 Conclusions

Specification, representation and enforcement of constraints has been a major factor of the impedance mismatch between object-oriented and data languages. In this paper we showed that interfacing object-oriented database technology with XML technology also stumbles on the problems of constraints, such as those available in XML Schema. Resolving the object-oriented/XML mismatch will be possible only in a constraint-based technology of the kind presented in this paper.

Object-oriented schemas that are based on object-oriented languages and their type systems cannot express the integrity constraints typical for either database schemas or dictated by the semantics of the application environment. The current underlying architecture of object-oriented assertion languages typically allows dynamic enforcement. We make two contributions that are relevant to transactions in particular.

The first contribution is to provide a virtual platform that integrates constraints with the run-time type system making the constraints available by reflection. This makes the integrity constraints of a schema visible by transactions and application programs in general. This platform allows a variety of constraint management and enforcement scenarios.

The second contribution is in the usage of a verification system to statically verify at least some integrity constraints. In fact, it is the combination of dynamic and static verification that that we applied to transactions to decrease the cost of dynamic checking of database integrity constraints.

The complexity of verification systems such as PVS requires development of proof strategies [6] specifically targeted to verification of object-oriented transactions and to interfacing with XML technology. A sample strategy is given in the paper as it applies to transaction verification. These tailored proof strategies and more friendly user interfaces are a key requirement in making these tools accessible to database programmers.

References

1. Alagić, S., Royer, M.: Genericity in Java: Persistent and database system implications. The VLDB Journal (2007), http://www.springerlink.com/content/a0671813x8p28724/
2. Alagić, S., Royer, M., Crews, D.: Temporal verification of Java-like classes. In: Proceedings of FTfJP 2006 (2006), http://www.disi.unige.it/person/AnconaD/FTfJP06/
3. Alagić, S., Royer, M., Briggs, D.: Program verification techniques for XML Schema-based technologies. In: Proceedings of ICSOFT, vol. 2, pp. 86–93 (2006)
4. Alagić, S., Royer, M.: Next generation of virtual platforms, http://www.odbms.org/experts.aspx#article4
5. Alagić, S., Logan, J.: Consistency of Java transactions. In: Lausen, G., Suciu, D. (eds.) DBPL 2003. LNCS, vol. 2921, pp. 71–89. Springer, Heidelberg (2004)
6. Archer, M., Di Vito, B., Munoz, C.: Developing user strategies in PVS: A tutorial. In: Proceedings of STRATA 2003 (2003)
7. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview, Microsoft Research 2004. Also in Proceedings of CASSIS 2004 (2004)
8. Benzaken, V., Doucet, D.: Themis: A database language handling integrity constraints. VLDB Journal 4, 493–517 (1994)
9. Benzanken, V., Schaefer, X.: Static integrity constraint management in object-oriented database programming languages via predicate transformers. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 60–84. Springer, Heidelberg (1997)
10. Cattell, R.G.G., Barry, D., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., Velez, F.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann, San Francisco (2000)
11. Document Object Model (DOM), http://www.w3.org/TR/REC-DOM-Level-1/
12. Fan, W., Simeon, J.: Integrity constraints for XML. Journal of Computer and System Sciences 66, 254–291 (2003)
13. Lammel, R., Meijer, E.: Revealing the X/O impedance mismatch. Microsoft Corporation (2007), http://homepages.cwi.nl/~ralf/xo-impedance-mismatch/paper.pdf
14. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cook, D., Muller, P., Kiniry, J.: JML Reference Manual, draft (July 2005), http://www.cs.iastate.edu/~leavens/JML/
15. Owre, S., Shankar, N., Rushby, J.M., Stringer-Clavert, D.W.J.: PVS Language Reference, SRI International, Computer Science Laboratory, Menlo Park, California
16. Owre, S., Shankar, N.: Writing PVS proof strategies, Computer Science Laboratory, SRI International, http://www.csl.sri.com
17. Royer, M., Alagić, S., Dillon, D.: Reflective constraint management for languages on virtual platforms. Journal of Object Technology 6, 59–79 (2007)
18. Sheard, T., Stemple, D.: Automatic verification of database transaction safety. ACM Transactions on Database Systems 14, 322–368 (1989)
19. Spelt, D., Even, S.: A theorem prover-based analysis tool for object-oriented databases. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 375–389. Springer, Heidelberg (1999)
20. LINQ to XSD, Microsoft (2007), http://blogs.msdn.com/xmlteam/archive/2006/11/27/typed-xml-programmer-welcome-to-linq.aspx
21. LINQ to XML, Microsoft (2006), http://www.xlinq.net/
22. XML Schema 1.1, http://www.w3.org/XML/Schema