Journal of Universal Computer Science, vol. 15, no. 1 (2009), 241-272
submitted: 18/1/08, accepted: 2/6/08, appeared: 1/1/09 © J.UCS

Formal Verification of Semistructured Data Models in PVS

Scott Uk-Jin Lee, Gillian Dobbie, Jing Sun
(The University of Auckland, New Zealand
{scott, gill, jing}@cs.auckland.ac.nz)

Lindsay Groves
(Victoria University of Wellington, New Zealand
lindsay@mcs.vuw.ac.nz)

Abstract: The rapid growth of the World Wide Web has resulted in a dramatic increase in semistructured data usage, creating a growing need for effective and efficient utilization of semistructured data. In order to verify the correctness of semistructured data design, precise descriptions of the schemas and transformations on the schemas must be established. One effective way to achieve this goal is through formal modeling and automated verification. This paper presents the first step towards this goal. In our approach, we have formally specified the semantics of the ORA-SS (Object-Relationship-Attribute data model for Semistructured data) data modeling language in PVS (Prototype Verification System) and provided automated verification support for both ORA-SS schemas and XML (Extensible Markup Language) data instances using the PVS theorem prover. This approach provides a solid basis for verifying algorithms that transform schemas for semistructured data.

Key Words: Semistructured data, Data modeling, Automated verification, ORA-SS, PVS.
Category: H.2.1, D.2.4

1 Introduction

As a result of the rapid growth of the World Wide Web and its technologies, many businesses and organizations are relying more on computers to carry out their business and services over the internet. While moving to online services has increased convenience and efficiency a great deal, the correctness and consistency of the data stored, retrieved, and updated by these services has become even more critical. For example, inconsistencies or corruption in data transactions containing account balances, confidential legal information, or credit card payments with online shopping, could lead to disastrous results.

Semistructured data is one of the well known standards for representing and exchanging data over the internet. The wide usage of semistructured data is not limited to Web applications, but includes various other applications such as digital libraries, biological databases, and multimedia data management systems [Wu et al., 2001]. In order to provide effective and efficient utilization of semistructured data, many researchers have proposed to design and develop database systems for semistructured data. As a result, several database systems have been developed for storing, manipulating, and querying semistructured data in the form of eXtensible Markup Language (XML) [Harold and Means, 2004], while traditional database companies, such as Oracle, have provided XML support for their existing database systems.

Consistency of data is even more important when the intention is to store the data long term. The data is initially stored with a particular meaning and this meaning must be enforced over time. Consider for example relational databases, where the meaning of the data is initially captured in ER (entity relationship) diagrams or similar, used in database design, and enforced through the design of the database and the schema. Algorithms that transform the schema, such as normalization, must ensure that none of the meaning is lost. The first step to guaranteeing that the meaning is not lost is to formally describe the data model and the semantics captured in the data model. In this paper we:

- informally describe the expected semantics of semistructured data
- formally capture the expected semantics of semistructured data
- provide a method for automatically verifying that a schema is correct with respect to the expected semantics
- provide a method to automatically validate an instance of semistructured data against a schema

Among many data modeling languages for semistructured data, we chose the Object Relationship Attribute model for Semistructured Data (ORA-SS) [Dobbie et al., 2001; Ling et al., 2005] because ORA-SS not only captures the constraints that are represented in textual languages such as XML Schema [Thompson et al., 2007] but also captures additional semantics that can be used for conceptual modeling. For formal verification, the Prototype Verification System (PVS) [Owre et al., 1992] was chosen because it has been used effectively to provide precise formal definitions and powerful automated verification support in various research projects [Vitt and Hooman, 1996; Srivas et al., 1997; Lawford and Wu, 2000].

Based on the semantics of ORA-SS, the standard representation for schema and data with their associated properties and constraints can be defined and verified in PVS. The relationship between an ORA-SS schema and its data instances can be specified to provide validation of the data against its schema. Our long term goal is depicted in Figure 1, which shows the overall structure of the verification process. The structure of the verification process presented in Figure 1 is divided into two parts by a dashed line, where the left hand side represents the part presented in this paper and the right hand side represents the part to be implemented in the future. We have published earlier work related to this project

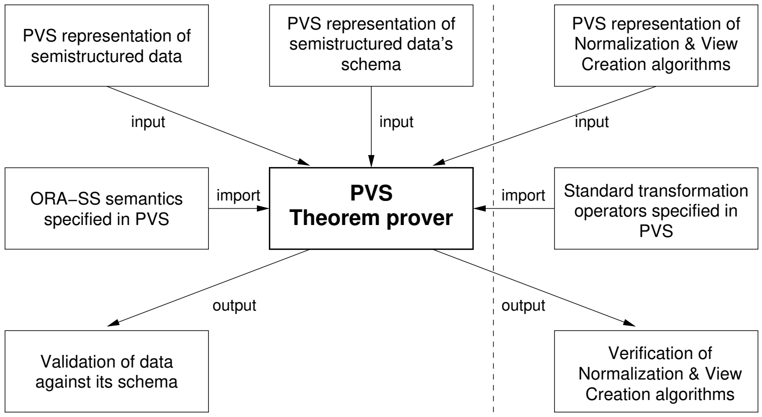

Figure 1: Verification support for semistructured data: schema, instance and normalization algorithm

in [Lee et al., 2005; Lee et al., 2006], and preliminary results of the future work in [Lee et al., 2008b; Lee et al., 2008a]. For the validation of the semistructured data against its schema, the PVS representation of semistructured data and its schema are input into the PVS theorem prover, together with the pre-defined ORA-SS semantics, and then the theorem prover can validate the data against its schema. In the future to verify the transformation algorithms, PVS representations of normalization and view creation algorithms will be passed into the theorem prover together with the defined library of transformation operators. PVS will then verify the algorithms to ensure the consistency of the data after such a transformation.

The rest of the paper is organized as follows. Section 2 presents background information about ORA-SS and PVS, which includes verification criteria of ORA-SS schemas and data instances and the PVS language and its theorem prover. In section 3, we present a formal semantics of the ORA-SS language and its data models in PVS with examples. Section 4 demonstrates the verification of ORA-SS schemas and data models using the PVS theorem prover. Section 5 compares this work with related work, and section 6 concludes the paper and addresses future work.

2 Background

This section provides an overview of the ORA-SS data modeling language, showing how ORA-SS models are described using ORA-SS diagrams and outlining the constraints that diagrams must satisfy in order to represent meaningful data models (further details will be presented in section 3). We also explain informally what it means for an XML document to be an instance of an ORA-SS schema, and provide a brief introduction to PVS.

2.1 ORA-SS Data Modeling Language

The Object-Relationship-Attribute model for Semistructured data (ORA-SS) is a semantically rich graphical data modeling language for semistructured data design [Dobbie et al., 2001]. It has been used in many XML related database applications [Ling et al., 2001; Ling et al., 2005]. ORA-SS models are based on four basic concepts: object classes, relationship types, attributes and references.

- An object class represents a type of an entity in the domain being modeled, such as person, student, or course. In ORA-SS diagrams, object classes are denoted by labeled rectangles.
- A relationship type represents a relationship among object classes. In semistructured data, this relationship is typically a nested relationship where an object class can relate to either an object class or a relationship type. In ORA-SS diagrams, a relationship type is described by an edge labeled with a tuple ($name$, n, p, c), where $name$ is the name of the relationship type, integer n indicates the degree of the relationship type which represents whether the object class is related to another object class or a relationship type, p represents the participation constraint of the parent object class in the relationship type and c represents the participation constraint of the child object class.
- An attribute represents a property of either an object class or a relationship type. In ORA-SS diagrams, attributes are denoted by labeled circles. An attribute may be an key attribute that has a unique value, and is represented as a filled circle. Other types of attributes include single valued attributes, multi-valued attributes, required attributes, and composite attributes.
- An object class can reference another object class indicating that the object class is extended with the details of the other object class. It is similar to the inheritance relationship in the object oriented data model. In ORA-SS diagrams, references are represented by dashed edges.

Figure 2 presents an ORA-SS schema diagram for a *Course-Student* data model. This ORA-SS schema diagram essentially consists of three tree structures (ignoring the dashed lines) describing various relationships, linked by references. The part of the diagram on the left represents the relationships between '*course*', '*prerequisite*', '*student*', and '*tutor*' object classes, where appropriate attributes

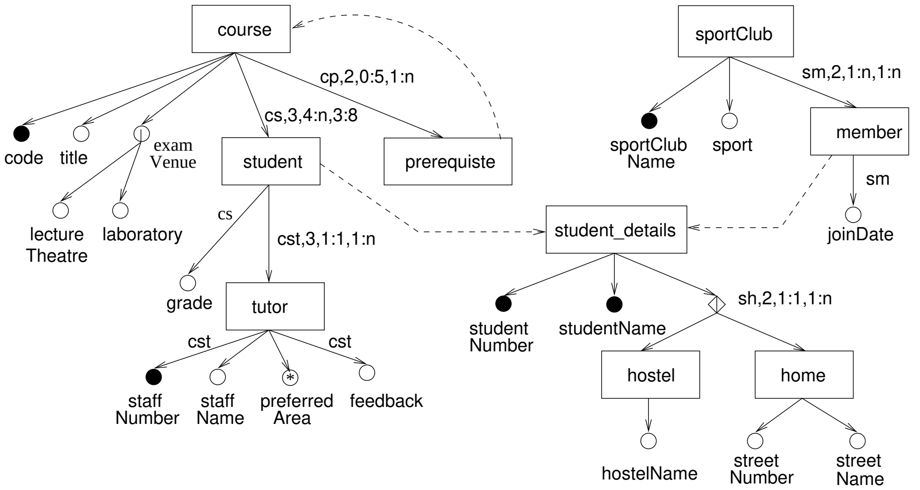

Figure 2: An ORA-SS schema diagram of a *Course-Student* data model.

are related to each object class to specify properties of the object classes. There are also several attributes, e.g., the '*grade*' of relationship '*cs*', describing properties associated with relationships. The attributes with filled circles, such as '*code*', represent the primary keys of object classes. The '*' in a circle indicates that the attribute can occur zero or more times, that is a '*tutor*' can have zero or more '*preferredAreas*'. The '|' in a circle represents a disjunctive attribute, so a '*course*' has an '*examVenue*' which is either a '*lectureTheatre*' or a '*laboratory*'. The relationship '*sm*' represents the membership of a student in a sport club, while the relationship '*sh*' describes a property of students. In fact '*sh*' is a disjunctive relationship which means that the object class can have a relationship with either of the object classes in the disjunctive relationship, i.e. a student can live in either a hostel or at home.

Note that the '*student_details*' object class is referenced by both the '*student*' object class and the '*member*' object class. Referencing provides properties of one object class to another object classes without duplicating the information. That is a '*member*' object class has the same properties as a '*student_details*' object class, with one additional property, '*joinDate*'. Similarly a prerequisite is simply a course, so there is a reference from object class '*prerequisite*' to object class '*course*'. In Figure 2, we have deliberately introduced three semantic errors for the purpose of illustrating the constraints discussed in sections 2.2 and 2.3, where we describe how they are detected.

2.2 Constraints on ORA-SS Schema Model

In order to define meaningful data models, an ORA-SS schema diagram must satisfy a number of additional constraints. For example:

- A relationship type r with degree two relates a parent object class with a child object class.
- In a relationship type r with degree n greater than two, there must be another relationship type with degree $n-1$, relating $n-1$ ancestors of the child object class of relationship r.
- A disjunctive relationship type relates a parent object class to two or more child object classes.
- A composite attribute or disjunctive attribute relates an attribute to two or more sub-attributes.
- A candidate key of an object class is a set of attributes selected from the attributes of the object class. A composite key of an object class is a set of attributes selected from two or more attributes of the object class.
- There can only be one primary key per object class, and it must be a candidate key of the object class.
- Relationship attributes have to belong to an existing relationship type.
- An object class can reference one object class only, but an object class can be referenced by multiple object classes.

These constraints can be used to check whether an ORA-SS data model is consistent. For example, examining the *Course-Student* schema diagram in Figure 2 reveals three semantic errors:

1. the degree of relationship '*cs*' between object class 'course' and 'student' is described as 3, representing a ternary relationship where it is actually binary, which violates the second constraint;
2. two attributes are described as primary keys for the object class '*student*', which violates the sixth constraint; and
3. the candidate key '*staffNumber*' is represented as an attribute of the relationship '*cst*', violating the fifth constraint which says that candidate keys can only be composed of attributes that belong to object classes.

In a small example like this, these errors can be easily detected by examining the schema diagram. However, examining large ORA-SS schema models for semistructured data can be error-prone — manually checking diagrams does not guarantee the consistency of the schema since it is likely that inconsistencies are not revealed when the schema structure is complicated. The formal specification of ORA-SS semantics presented in section 3 will be used in section 4 to detect inconsistencies such as these.

2.3 Constraints on ORA-SS Data Instance

In order for a semistructured data instance, such as an XML document, to conform to an ORA-SS schema, its structure must satisfy the constraints embodied in the ORA-SS schema. For example:

- Relationship instances must conform to their parent and child participation constraints, i.e., the number of child objects related to a single parent object or relationship instance should be consistent with the parent participation constraints; and the number of parent objects or relationship instances that a single child object relates to should be consistent with the child participation constraints.
- In a disjunctive relationship/attribute, only one object class/attribute can be associated to a particular parent instance.
- The value of a candidate key (single or composite) should uniquely identify an object in its object class.
- The number of values of a multi-valued attribute must be limited by the minimum and maximum cardinality values of the attribute.

A semistructured data instance can be represented as either a text-based XML document or a tree structured directed graph called an ORA-SS instance diagrams [Ling et al., 2005]. Figure 3 shows a semistructured data instance in the form of an XML document. This XML document does not conform to the ORA-SS schema in Figure 2, even after the semantic errors identified in the previous section are rectified. Inspecting this data instance, we can see that there are three inconsistencies:

1. the student in course '*compsci101*' is related to two tutors, violating the cardinality constraint of relationship '*cst*' which is defined as '*1:1*';
2. there are two identical values for the '*studentNumber*' primary key for two different students, violating the constraint that candidate keys and therefore primary keys must uniquely identify an object in an object class; and

```
<course>
  <code> compSci101 </code>
  ...
  <student idref = s123>
    <grade> A </grade>
    <tutor staffNumber = 186>
      <staffName> Scott Lee </staffName>
      <feedback> good </feedback>
    </tutor>
    <tutor staffNumber = 35>
      <staffName> John Smith </staffName>
      <feedback> ok </feedback>
    </tutor>
  </student>
  ...
</course>
...
```

```
<student_details studentNumber = 123, id = s123>
  <studentName> Kevin Jones </studentName>
  <hostel>
    <hostelName> Int. House </hostelName>
  </hostel>
  <home>
    <streetNumber> 3 </streetNumber>
    <streetName> Queen St </streetName>
  </home>
</student_details>
<student_details studentNumber = 123, id = s215>
  <studentName> Karl Thompson </studentName>
  <hostel>
    <hostelName> Orakei </hostelName>
  </hostel>
</student_details>
...
```

Figure 3: An XML data instance for a *Course-Student* data model.

3. the student '*KevinJones*' is related to both '*hostel*' and '*home*' where this is represented as a disjunctive relationship in the schema showing that a student can relate to either a hostel or a home, but not both.

One can see that it is not trivial to reveal the errors using only the schema diagrams and XML document. Consider examining a large database consisting of a number of XML documents that conform to a complicated ORA-SS schema model, it is almost impossible to verify all the data manually. Therefore, automated verification support based on formal specification can be beneficial. The automated verification support enables prompt and effective verification of a large number of data against the complicated ORA-SS schema models. In addition, it eliminates the possibility of undetected errors included in the result of a verification process.

2.4 Prototype Verification System (PVS)

PVS is a typed higher-order logic verification system, where a formal specification language is integrated with supporting tools [Owre et al., 1992]. It provides formal specification and verification through type checking and theorem proving. PVS has a number of language constructs including user-defined types, built-in types, functions, sets, tuples, records, enumerations, and recursively-defined data types such as lists and binary trees. Predicate subtypes and dependent types are included in PVS to introduce constraints in other types such as the type of prime numbers. These types that represent constraints increase the expressiveness and naturalness of the specifications, and the constraints on the types can be proved automatically by the theorem prover. With the provided language constructs,

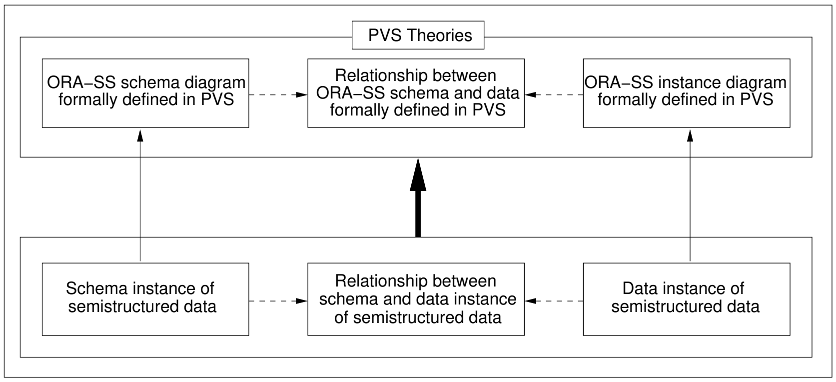

Figure 4: Structure of formally specified ORA-SS semantics and its verification model.

PVS specifications are represented in parameterized theories that contain assumptions, definitions, axioms, and theorems. Definitions of PVS also support recursive functions, inductively-defined relations and its expressions provide the usual arithmetic and logic operators, function application, lambda abstraction, and quantifiers. The definitions and expressions of PVS allow composition of complex specifications which lead to the easier construction of real world problems and the represented problems can be solved and verified using the built-in theories and theorem prover. PVS is considered one of the most popular and effective verification systems, because it provides an automated type checker, a rich library of predefined theorems and a powerful theorem prover. Many applications have adopted PVS to provide formal verification support to their system properties [Vitt and Hooman, 1996; Srivas et al., 1997; Lawford and Wu, 2000].

3 Formal Semantics of ORA-SS Data Model

In this section, we define the structure and semantics of ORA-SS data models and data instances using the PVS specification language. We have limited ourselves to a subset of the ORA-SS data model that demonstrates the major concepts in the language. Our approach consists of three theories as shown in Figure 4: one defining ORA-SS schemas, one defining data instances, and one defining the relationship between an instance and a schema. Specific ORA-SS schema diagrams and instances can then be translated into the vocabulary provided by these theories, to obtain PVS representations that can then be used for verification, as described in section 4.[1] Due to the space limitation, not all the semantic definitions for ORA-SS are presented here. A complete PVS semantics of ORA-SS can be found at `www.cs.auckland.ac.nz/~scott/files/JUCS/`.

3.1 The ORA-SS Schema Theory

We first define a PVS theory describing the structure and semantics of ORA-SS schemas. Every ORA-SS schema is built on a set of object class names and a set of attribute names, which (must both be non-empty and) are provided as arguments, OC and ATT, to a generic theory:

```
orassSchemaDef[OC, ATT: TYPE+]: THEORY
```

3.1.1 Schema Type

To capture the meaning of an ORA-SS schema diagram, we need to describe the relationships, attributes and keys embodied in the diagram. In our formal model, a schema is represented using a record type, with components representing these aspects of a diagram, with various kinds of attributes (object attributes, relationship attributes, composite attributes and disjunctive attributes) and keys (candidate keys and primary keys) represented separately:

```
Schema: TYPE =
  {s: [# relList: list[Relationship], oAttList: list[ObjectAtt],
  rAttList: list[RelationshipAtt], cAttList: list[CompositeAtt],
  dAttList: list[DisjunctiveAtt], cKeyList: list[CandidateKey],
  pKeyList: list[PrimaryKey]#] |
  noRelRepeat?(relList(s)) ∧
  noOAttRepeat?(oAttList(s)) ∧ noRAttRepeat?(rAttList(s)) ∧
  noCAttRepeat?(cAttList(s)) ∧ noDAttRepeat?(dAttList(s)) ∧
  noCKeyRepeat?(cKeyList(s)) ∧ noPKeyRepeat?(pKeyList(s)) ∧
  cKeyCorrect?(cKeyList(s), oAttList(s)) ∧
  pKeyCorrect?(pKeyList(s), cKeyList(s))}
```

The types of these components will be defined below. Note that we use lists to represent sets of relationships, attributes and keys, because these are processed more efficiently by PVS. The use of lists has introduced definitions of the functions such as '*noRelRepeat?*', '*noOAttRepeat?*' and '*noCKeyRepeat?*' for the record type to prevent the same components repeating multiple times in a list. In order for a schema to be valid, it must satisfy the additional constraints shown in the definition: there can be no repeated elements in the lists (because they are really intended to be sets), and keys must satisfy correctness constraints, which are described below.

[1] The standard PVS syntax [Owre et al., 1992] is used in specifying formal semantics of the ORA-SS language and examples.

3.1.2 Relationship Type

To describe a relationship type, we must record the object classes that participate in the relationship, the degree of the relationship, and the parent and child participation constraints. A relationship type is thus represented as a record with these four components:

$$
\begin{array}{l}
\text{Relationship}: \textsc{type} = \\
\quad \{r: [\#\ \text{rel}: \text{RelType},\ \text{degree}: \text{posnat}, \\
\quad \text{pConstraint}: [\text{nat},\ \text{posnat}],\ \text{cConstraint}: [\text{posnat},\ \text{posnat}]\ \#]\ \mid \\
\quad \text{degree}(r) = \text{length}(\text{rel}(r)) \wedge \\
\quad (\text{length}(\text{rel}(r)) > 2 \Rightarrow (\exists(\text{subRel}: \text{RelType}): \text{subRel} = \text{cdr}(\text{rel}(r))))\}
\end{array}
$$

The degree is declared as a positive integer (in fact, the degree must be at least 2) and the parent and child participation constraints are ordered pairs of numbers giving the minimum and maximum number of occurrences permitted. The minimum for parent participation is a natural number, since the parent objects in a relationship are not required to have any child objects. The others are all positive integers. The name of the relationship is not included in the type definition since it can be represented as a variable name of the relationship type in schema instances.

In this definition, '*rel*' lists the object classes participating in the relationship, starting with the one occurring deepest in the ORA-SS diagram, and working upwards. Its length must be the same as '*degree*'. We define '*rel*' as a list of sets of object classes, and use sets containing more than one object class to represent disjunctive relationships. Since a relationship must involve at least object classes, the list must contain at least two elements, and since relationships must not be circular, the list must not contain cycles:

$$
\begin{array}{l}
\text{RelType}: \textsc{type} = \\
\quad \{\text{ocSetList}: \text{list}[\text{set}[\text{OC}]]\ \mid \\
\quad \text{length}(\text{ocSetList}) > 1 \wedge \text{no\_cycle?}(\text{ocSetList})\}
\end{array}
$$

$$
\begin{array}{l}
\text{no\_cycle?}(\text{ocSetList}: \text{list}[\text{set}[\text{OC}]]): \textsc{recursive}\ \text{bool} = \\
\quad \textsc{cases}\ \text{ocSetList}\ \textsc{of} \\
\quad\quad \text{null}: \textsc{true}, \\
\quad\quad \text{cons}(\text{head},\ \text{tail}): (\forall(\text{ocSet}: \text{set}[\text{OC}]): \text{member}(\text{ocSet},\ \text{tail}) \Rightarrow \\
\quad\quad\quad\quad \text{disjoint?}(\text{head},\ \text{ocSet})) \wedge \text{no\_cycle?}(\text{tail}) \\
\quad \textsc{endcases} \\
\textsc{measure}\ \text{length}(\text{ocSetList})
\end{array}
$$

The definition of '*Relationship*' requires that if a relationship has degree n greater than two, then there is also a relationship with degree $n-1$ relating the ancestors of the child object class of the relationship in the same hierarchical order. This ensures that a relationship has the right number of participants.

3.1.3 Attributes of Object Class and Relationship

In ORA-SS schema diagrams, object classes and relationships can both have attributes, and these attributes may be composite or disjunctive. The set of attributes for an object class is defined as a record with two components, giving the object class (*oc*) and its attributes (*attList*); similarly for attributes of relationships. As usual, there can be no repeated attributes.

```
ObjectAtt: TYPE =
  {oAtt: [# oc: OC, attList: list[ATT] #] |
  noAttRepeat?(attList(oAtt))}

RelationshipAtt: TYPE =
  {rAtt: [# rel: RelType, attList: list[ATT] #] |
  noAttRepeat?(attList(rAtt))}
```

Composite attributes are defined using the '*CompositeAtt*' record type which has two components giving the attribute (*att*) and its component attributes (*attList*).

```
CompositeAtt: TYPE =
  {cAtt: [# att: ATT, attList: list[ATT]#] |
   length(attList(cAtt)) > 1 ∧ noAttRepeat?(attList(cAtt)) ∧
   ¬ member(att(cAtt), attList(cAtt))}
```

A composite attribute must have at least two components and must not have repeated components. Also, an attribute must not be listed as one of its own components. Disjunctive attributes are defined in a similar way with similar type constraints.

3.1.4 Candidate Key and Primary Key

In ORA-SS schema diagrams, an object class may have any number of sets of attributes, called *candidate keys*, each having a unique value for each instance of that object class. A candidate key consisting of more than one attribute is called a *composite candidate key.* The set of candidate keys for an object class is defined as a record with two components, giving the object class (*oc*) and its candidate keys (*keyList*).

```
CandidateKeys: TYPE =
  {cKey: [# oc: OC, keyList: list[list[ATT]] #] |
   (∀(attList: list[ATT]): member(attList, keyList(cKey)) ⇒
    noAttRepeat?(attList)) ∧ noKeyRepeat?(keyList(cKey))}
```

The recursive predicate function '*cKeyCorrect?*' is defined for the candidate key definition to apply constraints of ORA-SS semantics on a candidate key. The semantics of ORA-SS specifies that a candidate key should be selected from the set of attributes that belong to the object class, hence the '*cKeyCorrect?*' function checks whether the entire candidate keys for all object classes satisfy this condition using the '*correctOAtt4CKey?*' function. The '*correctOAtt4CKey?*' function uses various other functions to provide the constraint checking for the candidate keys.

```
cKeyCorrect?(cKeyList: list[CandidateKeys], oAttList: list[ObjectAtt]):
RECURSIVE bool =
  CASES cKeyList OF
    null: TRUE,
    cons(head, tail): correctOAtt4CKey?(oAttList, head) ∧
                        cKeyCorrect?(tail, oAttList)
  ENDCASES
MEASURE length(cKeyList)
```

In ORA-SS schema diagrams, an object class has a primary key which is selected from the set of candidate keys. The primary key is defined similarly as a record type that consists of an object class and list of attributes. The list of attribute refers to the primary key or composite primary key that belongs to the object. Also there is a recursive predicate function that checks whether a primary key is selected from candidate keys as specified in ORA-SS semantics.

3.2 Representing a Schema Diagram in PVS

To illustrate how a ORA-SS schema diagram can be represented using the definitions presented above, we will show how, the *Course-Student* schema diagram in Figure 2 can be represented in a PVS theory called '*schemaEx*' as follows.

```
schemaEx: THEORY
  BEGIN

  IMPORTING OC
  IMPORTING ATT
  IMPORTING orassSchemaDef[OC, ATT]

  cp: RelType = (:singleton(prerequisite), singleton(course):)

  cs: RelType = (:singleton(student), singleton(course):)
  ...
  cpRel: Relationship =
    (# rel := cp, degree := 2,
    pConstraint := (0, 5), cConstraint := (1, many) #)

  csRel: Relationship =
    (# rel := cs, degree := 3,
    pConstraint := (4, many), cConstraint := (3, 8) #)
  ...
  courseAtt: ObjectAtt =
    (# oc := course, attList := (:code, title, examVenue:) #)
```

```
tutorAtt: ObjectAtt =
  (# oc := tutor, attList := (: staffName, preferredArea:) #)
...
csRelAtt: RelationshipAtt =
  (# rel := cs, attList := (:grade:) #)

cstRelAtt: RelationshipAtt =
  (# rel := cst, attList := (:staffNumber, feedback:) #)
...
examVenueDAtt: DisjunctiveAtt =
  (# att := examVenue, attList := (:lectureTheatre, laboratory:) #)

courseCKey: CandidateKey =
  (# oc := course, keyList := (:(:code:):) #)

tutorCKey: CandidateKey =
  (# oc := tutor, keyList := (:(:staffNumber:):) #)
...
coursePKey: PrimaryKey = (# oc := course, keyList := (:code:) #)

tutorPKey: PrimaryKey = (# oc := tutor, keyList := (:staffNumber:) #)
...
courseStudent: Schema =
  (# relList := (:cpRel, csRel, cstRel, shRel, smRel:),
  oAttList := (:courseAtt, tutorAtt, studentAtt,  ...  sportClubAtt:),
  rAttList := (:csRelAtt, cstRelAtt, smRelAtt:),
  cAttList := null, dAttList := (:examVenueDAtt:),
  cKeyList := (:courseCKey, tutorCKey, studentCKey, sportClubCKey:),
  pKeyList := (:coursePKey, tutorPKey,  ...  sportClubPKey:) #)

END schemaEx
```

The '*schemaEx*' theory imports the *OC* and *ATT* data types constructed for the *Course-Student* schema, and the '*orassSchemaDef*' theory. It then defines several local variables holding the components for the *Course-Student* schema, as shown in the diagram in Figure 2, and finally defines *courseStudent* as a schema, using these local variables. The complete *Student-Course* representation in '*SchemaEx*' theory is defined in `schemaEx.pvs`.

3.3 The ORA-SS Data Instance Theory

An ORA-SS data instance is an abstract representation of an XML document. The general structure and properties of ORA-SS data instances are constructed on a non-empty sets of object classes (*OC*), attributes (*ATT*), objects (*OBJECT*) and attribute values (*ATTVALUE*), which are provided as arguments to a generic theory:

```
orassDataDef[OC, OBJECT, ATT, ATTVALUE: TYPE+]: THEORY
```

To capture the meaning of an ORA-SS data instance, we need to describe the instances of relationships and instances of attributes. The data instance is defined using record type, with object relationships (*oRelList*), object tree path (*oRelTreeList*) representing relationship instances, and with attribute values (*attInstList*, *cAttValList* and *dAttValList*) representing attribute instances. The types of these components will be described in the following sub-sections. The additional constraints of the above definition specifies that there can be no repeated elements in the list of each components.

```
Data: TYPE =
  {d: [# oRelList: list[ObjRelationship], oRelTreeList: list[ObjRelTree],
  attInstList: list[AttInstance], cAttValList: list[CompositeAttVal],
  dAttValList: list[DisjunctiveAttVal] #] |
  noORelRepeat?(oRelList(d)) ∧ noORelTreeRepeat?(oRelTreeList(d)) ∧
  attInstRepeat?(attInstList(d)) ∧ cAttValRepeat?(cAttValList(d)) ∧
  dAttValRepeat?(dAttValList(d))}
```

A complete PVS semantics of the constructs for data is defined in `orassDataDef.pvs`.

3.3.1 Basic Types

In ORA-SS data instance diagram, an instance of object class and an instance of attribute are described by an object with its object class and an attribute value with its attribute respectively. In our formal model, an instance of object class is represented using a record type, with components representing object class and its instance (object). An instance of attribute is represented with components representing attribute and its instance (attribute value). These basic types are defined initially to construct a formal specification of ORA-SS semantics for a data instance.

```
OBJ: TYPE+ = [# class: OC, object: OBJECT #]

ATTVAL: TYPE+ = [# attribute: ATT, value: ATTVALUE #]
```

3.3.2 Object Relationships

To describe an instance of relationship, the relationship between objects and the tree structure of the related objects must be recorded. Thus two PVS types representing the above two aspects in the ORA-SS data instance are both defined as a list of objects.

```
ObjRelationship: TYPE = {oList: list[OBJ] | length(oList) = 2}

ObjRelTree: TYPE = {oList: list[OBJ] | noObjCycle?(oList)}
```

The '*ObjRelationship*' type represents the relationship between objects where its constraint specifies that the object relationship in the list is always a binary.

This constraint is applied to the type since n-ary relationship is not possible between objects in a ORA-SS data instance. The '*ObjRelTree*' type are defined to represent the structural information (tree path) of the instance diagram as a list of objects where the constraints specifies that there is no repeated objects in the list. This definition is essential for reconstructing the data instance diagram from the PVS representation of the diagram.

3.3.3 Instances of Attributes, Composite Attributes, and Disjunctive Attributes

In ORA-SS data instance diagrams, instances of relationship attributes and object attributes are both represented as attribute values that belong to objects. Whereas, the instances of composite attribute and disjunctive attribute similar to their representation in a schema diagram. The instances of attribute for both object class and relationship are defined as a single record with two components, giving the objects (*objList*) and the attribute values (*attValList*) that belongs to them. The instances of composite and disjunctive attributes are defined as a record with two components, giving the attribute value and list attribute values for the instances of composite attributes; giving two attribute value for the instances of disjunctive attributes. As usual, there can be no repeated attributes.

```
AttInstance: TYPE = [# objList: list[OBJ], attValList: list[ATTVAL] #]

CompositeAttVal: TYPE =
  {cAttVal: [# attVal: ATTVAL, attValList: list[ATTVAL] #] |
   noAttValRepeat?(attValList(cAttVal))}

DisjunctiveAttVal: TYPE = [# attVal1: ATTVAL, attVal2: ATTVAL #]
```

In the '*AttInstance*' type definition, the list of objects is used to specify the exact object in the tree structure because the same object can appear many times in different tree paths. The '*DisjunctiveAttVal*' type is defined similarly to the definition for disjunctive attributes in the ORA-SS schema diagram. However, attribute value is used instead of list of attribute values because only one attribute can be selected from a set of disjunctive attributes.

3.4 Representing a Data Instance in PVS

To illustrate how an ORA-SS data instance can be represented using the definitions presented above, we will show how, the XML data instance for '*Course-Student*' schema in Figure 3 can be represented in a PVS theory called '*dataEx*' as follows. The '*dataEx*' theory imports the *OC*, *ATT*, *OBJECT* and *ATTVALUE* data types constructed for the instance of the *Course-Student* schema, and the '*orassDataDef*' theory. It then defines several local variables holding the components for the instance of *Course-Student* schema, as shown in the diagram in

Figure 3, and finally defines '*courseStudentInst*' as a data instance, using these local variables.

```
dataEx: THEORY
  BEGIN

  IMPORTING OC
  ...
  IMPORTING orassDataDef[OC, OBJECT, ATT, ATTVALUE]

  course1Obj: OBJ =
    (# class := course, object := course1 #)

  student1Obj: OBJ =
    (# class := student, object := student1 #)
  ...
  compSci101AV: ATTVAL =
    (# attribute := code, value := compSci101 #)

  examVenue101AV: ATTVAL =
    (# attribute := examVenue, value := examVenue101 #)
  ...
  cs1: ObjRelationship = (:student1Obj, course1Obj:)

  st1: ObjRelationship = (:tutor1Obj, student1Obj:)
  ..
  cstTree1: ObjRelTree = (:tutor1Obj, student1Obj, course1Obj:)

  cstTree2: ObjRelTree = (:tutor1Obj, student2Obj, course1Obj:)
  ..
  c1attInst: AttInstance =
    (# objList := (:course1Obj:),
    attValList := (:compSci101AV, pOPAV, examVenue101AV:) #)

  c2attInst: AttInstance =
    (# objList := (:course2Obj:),
    attValList := (:compSci105AV, pOCSAV, examVenue105AV:) #)
  ...
  eV101dAttVal: DisjunctiveAttVal =
    (# attVal1 := examVenue101AV, attVal2 := plt2AV #)
  ...
  courseStudentInst: Data =
    (# oRelList := (:cs1, cs2, cs3, cs4, cs5, cs6, cs7,  ...  sm2:),
    oRelTreeList := (:cstTree1, cstTree2, cstTree3, cstTree4,  ...  smTree2:),
    attInstList := (:c1attInst, c2attInst, c3attInst,  ...  s1m2attInst:),
    cAttValList := null,
    dAttValList := (:eV101dAttVal, eV105dAttVal, eV111dAttVal:) #)
```

A complete *Student-Course* instance representation in '*dataEx*' theory is defined in `dataEx.pvs`.

3.5 Relating ORA-SS Schemas and Data Instances

With the semantics of the ORA-SS schema diagram and data instance defined, we can specify the generic constraints of the mappings between an ORA-SS schema diagram and its data instance in PVS as shown below. The mappings are defined in a generic theory with the arguments of *OC*, *OBJECT*, *ATT*, and *ATTVALUE*.

```
orassSchemaNDataDef[OC, OBJECT, ATT, ATTVALUE: TYPE+]: THEORY
```

The PVS theories previously defined to describe the semantics of ORA-SS schema diagram and data instance (*orassSchemaDef* and *orassDataDef*) are imported to define the mappings between ORA-SS schema and its data instance.

```
IMPORTING orassSchemaDef[OC, ATT]
IMPORTING orassDataDef[OC, OBJECT, ATT, ATTVALUE]
```

3.5.1 Instances of Relationships

In ORA-SS data instance diagrams, instances of relationships are defined very differently to the definition of relationships for the ORA-SS schema. The instance of relationship are defined as list of binary object relationships and list of objects representing the structure of the objects in the data instance. The mapping for relationship and its instances is defined as a record type that consists of a relationship and a list of list of object relationships. The inner list is used to represent the ordered collection of binary object relationships that make up the instance of binary, ternary, and n-ary relationships. The outer list is used to represent the set of relationship instances.

```
RelInstance: TYPE =
  {rInst: [# rel: RelType, relInst: list[list[ObjRelationship]] #] |
  isRelInstAll?(toObjListAll(relInst(rInst)), rel(rInst)) ∧
  noRepeatingObjRel?(toObjListAll(relInst(rInst)))}
```

The additional constraints of relationship instance specifies that the object in the object relationships is an instance of corresponding object class in the relationship. The type constrains of '*RelInstance*'type transforms each list of '*ObjRelationship*' into a list of objects where each object corresponds to an object class in its relationship and checks whether each object in the list is an instance of the object class in the relationship. The other constraint of '*RelInstance*' record type specifies that there should not be any repeated relationship instances in the outer list.

3.5.2 Cardinalities of Relationships and Relationship Instances

In the ORA-SS data model, a set of relationship instances must satisfy the specified participation constraints of the corresponding relationship in the ORA-SS schema diagram. The predicate function called '*correctConstraints?*' is defined to verify whether the relationship instances satisfies both parent and child participation constraints.

```
parentSet(objListList: list[list[OBJ]], loParent: list[OBJ]): RECURSIVE nat =
  CASES objListList OF
    null: 0,
    cons(head, tail): (IF (¬(head = null) ∧ (¬loParent = null) ∧
                      loEqual?(cdr(head), cdr(loParent))) THEN 1
                      ELSE 0 ENDIF) + parentSet(tail, loParent)
  ENDCASES
MEASURE length(objListList)

correctPC?(oRelListList: list[list[ObjRelationship]], objListList: list[list[OBJ]],
pConst: [nat, posnat]): RECURSIVE bool =
  CASES oRelListList OF
    null: TRUE,
    cons(head, tail):
      (PROJ_1(pConst) ≤ parentSet(objListList, toObjList(head))) ∧
      (PROJ_2(pConst) ≥ parentSet(objListList, toObjList(head))) ∧
      correctPC?(tail, objListList, pConst)
  ENDCASES
MEASURE length(oRelListList)

correctConstraints?(r: Relationship, oRelListList: list[list[ObjRelationship]]):
bool =
  IF (oRelListList = null) THEN (PROJ_1(pConstraint(r)) = 0)
  ELSE (correctPC?(oRelListList, toObjListAll(oRelListList), pConstraint(r)) ∧
       correctCC?(oRelListList, toObjListAll(oRelListList), cConstraint(r)))
  ENDIF
```

The predicate function '*correctConstraints?*' uses a recursive predicate function '*correctPC?*' and '*correctCC?*' to verify whether the relationship instances satisfy the participation constraints of the relationship. The '*correctPC?*' function checks whether the number of relationship instances with the same parent object or parent object relationship is between '*min:max*' notation of parent participation constraint of the relationship for the entire relationships and its instances. The '*parentSet*' function is defined and used in '*corrrectPC?*' function to calculate the number of relationship instance with the same parent object or parent object relationship existing in a list of relationship instance for a relationship. The verification for child participation constraint for relationships is defined very similarly.

3.5.3 Instances of Object Attributes and Relationship Attributes

In the ORA-SS model, schema diagrams clearly distinguish attributes of object classes and attribute of relationship types, but data instance diagrams does not distinguishes between these two attribute instances. By examining the schema and data instance, instances of object attributes and instances of relationship attributes can be distinguished and defined for the mappings between schema and data.

```
ObjAttInstance: TYPE = [# obj: OBJ, objAttInst: list[ATTVAL] #]

RelAttInstance: TYPE = [# relInst: list[OBJ], relAttInst: list[ATTVAL] #]

correctAttInst?(attValList: list[ATTVAL], attList: list[ATT]): bool =
  IF (attList = null) THEN FALSE
  ELSE (list2set(attVal2Att(attValList)) ⊆ list2set(attList)) ENDIF
```

The instances of object attributes and the instances of relationship attributes are distinguished using record types '*ObjAttInstance*' and '*RelAttInstance*' respectively. The '*ObjAttInstance*' record type consists of an object and list of attribute values representing the set of attribute values that belong to the object. The '*RelAttInstance*' record type consists of a list of objects and list of attribute values representing the set of attribute values that belong to the relationship instance. There is also a '*correctAttInst?*' predicate function that checks whether the list attribute values is a correct instance of list of attributes. This predicate function can be used for both object or relationship attribute instances since both record types represent attribute instances as a list of attribute values and attribute as a list of attributes.

3.5.4 Instances of Candidate Keys

In the ORA-SS model, a value of a candidate key of an object instance is defined to be unique. In the definition of mappings between schema and data, verification of this property must be provided. For this verification the '*correctCKey?*' predicate function is defined to check uniqueness of each attribute value for a given candidate key.

```
correctCKey?(attListList: list[list[ATT]],
             attValList1, attValList2: list[ATTVAL]):
RECURSIVE bool =
  CASES attListList OF
    null: TRUE,
    cons(head, tail): ¬(attValListEqual?(findCKeyInst(head, attValList1),
                      findCKeyInst(head, attValList2))) ∧
                      correctCKey?(tail, attValList1, attValList2)
  ENDCASES
MEASURE length(attListList)
```

The '*correctCKey?*' recursive predicate function takes in the candidate key and two object attributes to check whether the values of the candidate key that belong to the two different objects is unique or not for all combinations of objects. The recursive function '*findCKeyInst*' is needed to find the instances of candidate key among the object attribute instances because ORA-SS data instance diagrams do not contain any information about candidate keys. A primary key and its instances inherit this property of uniqueness from candidate key since primary keys are selected from the candidate keys. Hence, a separate predicate function for checking uniqueness of primary key values is not necessary.

3.5.5 Mappings between Schemas and Data Instances

Similarly to '*Schema*' type and '*Data*' type definition, the entire mappings between schema and data is also defined as a single record type called '*SchemaData*'.

```
SchemaData: TYPE =
  {sd: [# relInstList: list[RelInstance], oAttInstList: list[ObjAttInstance],
  rAttInstList: list[RelAttInstance] #] |
  noRelInstRepeat?(relInstList(sd)) ∧ noOAttInstRepeat?(oAttInstList(sd)) ∧
  noRAttInstRepeat?(rAttInstList(sd)) ∧
  disjointRelInst?(toObjListAll(allRelInst?(relInstList(sd))))}
```

The '*SchemaData*' type is defined as a record type that consists of entire mappings between relationships, attributes, keys and their instances. Similarly to schema type and data type, list is used and there are type constraints to check whether each of these lists has any repeating elements in it. Also there is a extra constraint for mappings between a relationship and its instances that checks whether a set of instances for a relationship is disjoint from a set of instances for a different relationship. This constraint for checking disjointness of relationship instances is necessary because the instance of a relationship can only belong to a single relationship type.

3.5.6 Data Validation against Schema

In the '*SchemaNDataDef*' theory, data can be validated against its schema using the mappings between schema and data defined. This data validation against the schema is defined using a predicate function called '*correctSchemaNData?*' with '*Schema*' type, '*Data*' type, and '*SchemaData*' type as its arguments.

```
correctSchemaNData?(s: Schema, d: Data, sd: SchemaData): bool =
  relInstCorrectConstraints?(relList(s), relInstList(sd)) ∧
  oAttInstCorrect?(oAttInstList(sd), oAttList(s)) ∧
  rAttInstCorrect?(rAttInstList(sd), rAttList(s)) ∧
  cAttValCorrect?(cAttValList(d), cAttList(s)) ∧
  dAttValCorrect?(dAttValList(d), dAttList(s)) ∧
  cKeyCorrect?(cKeyList(s), oAttInstList(sd))
```

The '*correctSchemaNData?*' predicate function checks whether every relationship instance meets the participation constraints of the relationship using '*relInstCorrectConstraints?*', which utilizes '*correctConstraints?*' function. The predicate function also provides validation of attribute instances against the object attributes, relationship attributes, composite attributes, and disjunctive attributes. All the functions that checks for validity of the attribute instances uses the '*correctAttinst?*' function mentioned earlier. Finally, the predicate function checks the uniqueness of each candidate key instance for every object class using '*cKeyCorrect?*' that utilizes the '*correctCKey?*' function. When all these predicate functions in '*correctSchemaNData?*' are verified the data is validated against its schema. A complete PVS semantics of the ORA-SS constructs are defined in `SchemaNDataDef.pvs`.

3.6 Representing Mappings between Schema Diagrams and Data Instances in PVS

To illustrate how a mapping between a schema diagram and its data instance can be represented using the definitions presented above. We will show how the mappings between the '*Course-Student*' schema diagram in Figure 2 and its XML data instance in Figure 3 can be represented in a PVS theory called '*schemaNData*' as follows. The representation of the mappings imports the '*schemaEx*', '*dataEx*', and '*orassSchemaNDataDef*' theory to provide formal specification for the mappings between the schema diagram example and XML data example according to the mapping definition. It then defines several local variables holding the components for the mappings and finally defines *courseStudentMap* as a entire mappings between '*schemaEx*', '*dataEx*', using these local variables. The '*schemaNData*' relationship representation also includes a conjecture called '*correctSchemaNData_con*' which can be used to validate the '*dataEx*' against the '*schemaEx*' by verifying the conjecture.

```
schemaNdata: THEORY
  BEGIN

  IMPORTING schemaEx
  IMPORTING dataEx
  IMPORTING orassSchemaNDataDef[OC, OBJECT, ATT, ATTVALUE]

  cpRelInst: RelInstance =
    (# rel := cp, relInst := null #)

  csRelInst: RelInstance =
    (# rel := cs, relInst := (:(:cs1:), (:cs2:), (:cs3:), (:cs4:), (:cs5:),
    (:cs6:), (:cs7:), (:cs8:), (:cs9:), (:cs10:), (:cs11:), (:cs12:):) #)
  ...
  c1OAttInst: ObjAttInstance =
    (# obj := course1Obj,
```

```
    objAttInst := (:compSci101AV, pOPAV, examVenue101AV:) #)

  c2OAttInst: ObjAttInstance =
    (# obj := course2Obj,
    objAttInst := (:compSci105AV, pOCSAV, examVenue105AV:) #)
  ...
  c1s1RAttInst: RelAttInstance =
    (# relInst := (:student1Obj, course1Obj:),
    relAttInst := (:aAV:) #)

c1s2RAttInst: RelAttInstance =
  (# relInst := (:student2Obj, course1Obj:),
    relAttInst := (:bAV:) #)
...
courseStudentMap: SchemaData =
  (# relInstList := (:cpRelInst, csRelInst, cstRelInst, shRelInst, smRelInst:),
  oAttInstList := (:c1OAttInst, c2OAttInst, c3OAttInst, ... sp1OAttInst:),
  rAttInstList := (:c1s1RAttInst, c1s2RAttInst, ... s1m2RAttInst:) #)

correctSchemaNData_con: CONJECTURE
  correctSchemaNData?(courseStudent, courseStudentInst, courseStudentMap)
```

A complete representation for the mappings is defined in `schemaNdata.pvs`.

4 Formal Verification of ORA-SS Data Model

Using the formally defined ORA-SS semantics, we can perform automatic verification on both schema diagrams and data instances via the PVS theorem prover. For example, the *Course-Student* schema example in Figure 2 can be verified against the ORA-SS semantics, and the XML data instance of *Course-Student* schema in Figure 3 can be validated against its schema. In this section, we demonstrate that the pre-introduced errors in both Figure 2 and Figure 3 can be successfully detected by PVS. The verification and validation will be conducted using the type checker and theorem prover of PVS. For each theory, the PVS type checker uses the theorem prover to automatically prove all the Type Correctness Conditions (TCCs), which refers to type constraints applied to various types in the theories. If there is a complicated type constraint that cannot be verified automatically by the theorem prover, the type checker returns unproved TCCs. These unproved TCCs can be proved interactively using the predefined theorems and lemmas of PVS with little effort. After type checking the theories, the conjectures in the theories must be proven to complete the verification and validation. Similar to the proving of unproven TCCs, the conjectures can also be proven interactively. Once all the theories are type checked and all the conjectures are proven, verification of the schema instance and validation of the data instance are complete proving that the schema is consistent with respect to the ORA-SS semantics and the data is consistent with respect to its schema.

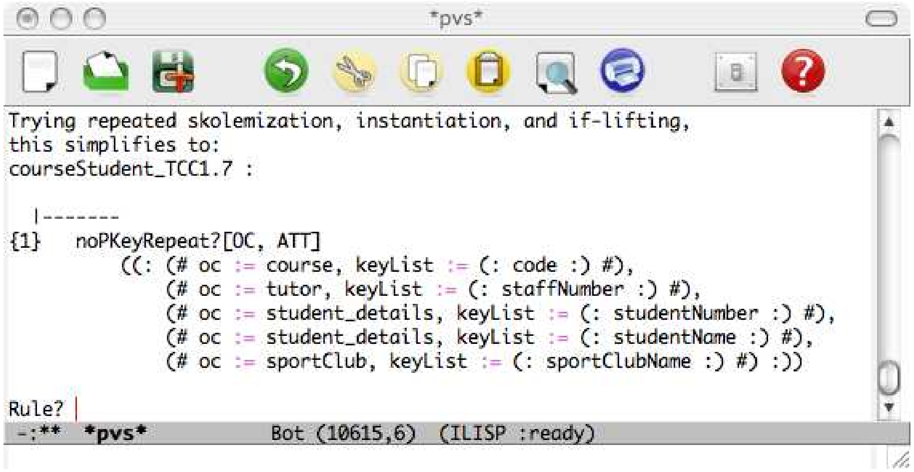

Figure 5: Verifying the type correctness condition for primary key

4.1 Formal Verification of ORA-SS Schema Diagram

We demonstrate the verification of the *Course-Student* schema diagram in Figure 2 as follows. This verification is conducted through type checking because all the constraints for the schema definition in the ORA-SS semantics are applied using type constraints. The '*schemaEx*' theory is verified to be consistent to the semantics of ORA-SS only if the type checking and theorem prover proves all the Type Correctness Conditions (TCCs).

The type checking of the '*schemaEx*' theory produces several unproved Type Correctness Conditions (TCCs) that must be proved. When the TCC for relationship '*cs*' is verified using the theorem prover, it results in a state, '$3 = 2$', which is improvable. This shows that the definition of cs degree is incorrect unless the '$3 = 2$' clause is proved.

After correcting the incorrect representation, type checking the *schemaEx*' theory again returned several TCCs. When the TCCs for schema type '*courseStudent*' is proved, it results in an improvable state as shown in Figure 5. The figure shows that the recursive predicate function '*noPKeyRepeat?*' cannot be proved after a manual attempt of verification. The '*noPKeyRepeat?*' function checks whether there is more than one primary key for a single object class or not. In this case, Figure 5 indicates that there are two primary keys defined for '*student_details*' object class.

After correcting the incorrect representation, type checking the '*schemaEx*' theory again returned several TCCs. When the TCCs for schema type '*courseStudent*' is proved, it results in an improvable state as shown in Figure 6.

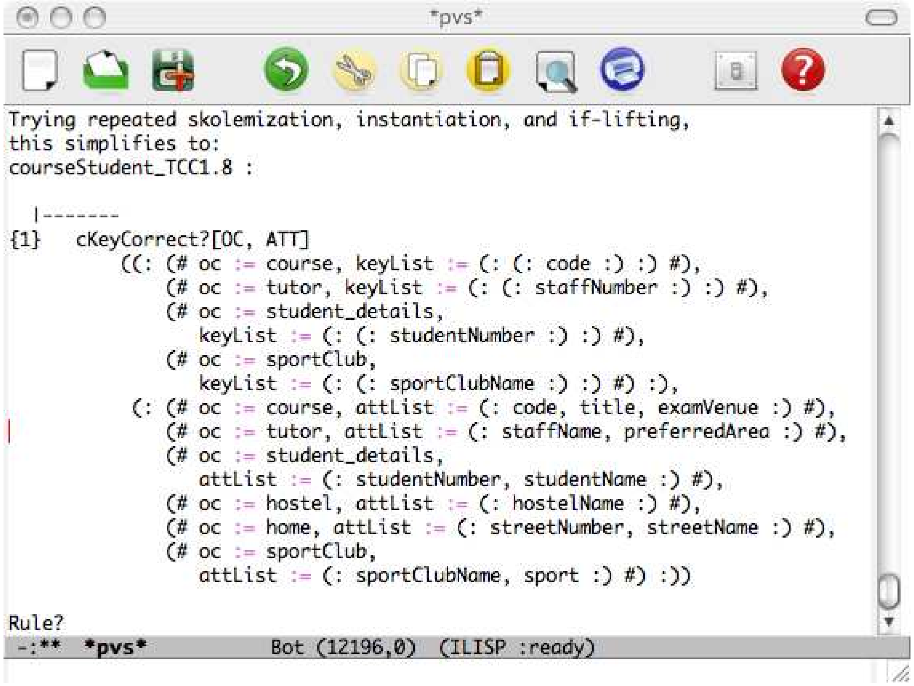

Figure 6: Verifying the type correctness condition for candidate key

The figure shows that the recursive predicate function '*cKeyCorrect?*' cannot be proved. The '*cKeyCorrect?*' function checks whether all the attributes in the candidate key are chosen from the objects attribute. In this case, Figure 6 shows that candidate key '*staffNumber*' for '*tutor*' object class does not belong to object class '*tutor*' as an attribute of the object class. When all three introduced errors are corrected and all the TCCs for the '*schemaEx*' theory are verified, the schema diagram example in Figure 2 is verified to be semantically correct.

4.2 Formal Validation of ORA-SS Data Instance

We demonstrate the validation of the XML data instance in Figure 3 as follows. This validation will be conducted through the type checking and verification of predicate functions represented as conjectures. The PVS representation for the XML data example and mappings between the schema diagram example and the XML data example must be verified to prove whether the definitions satisfy the type constraints applied. Then the '*correctSchemaNData_con*' conjecture in '*schemaNData*' theory must be verified through the theorem to prove the predicate function that checks the data validity against its schema.

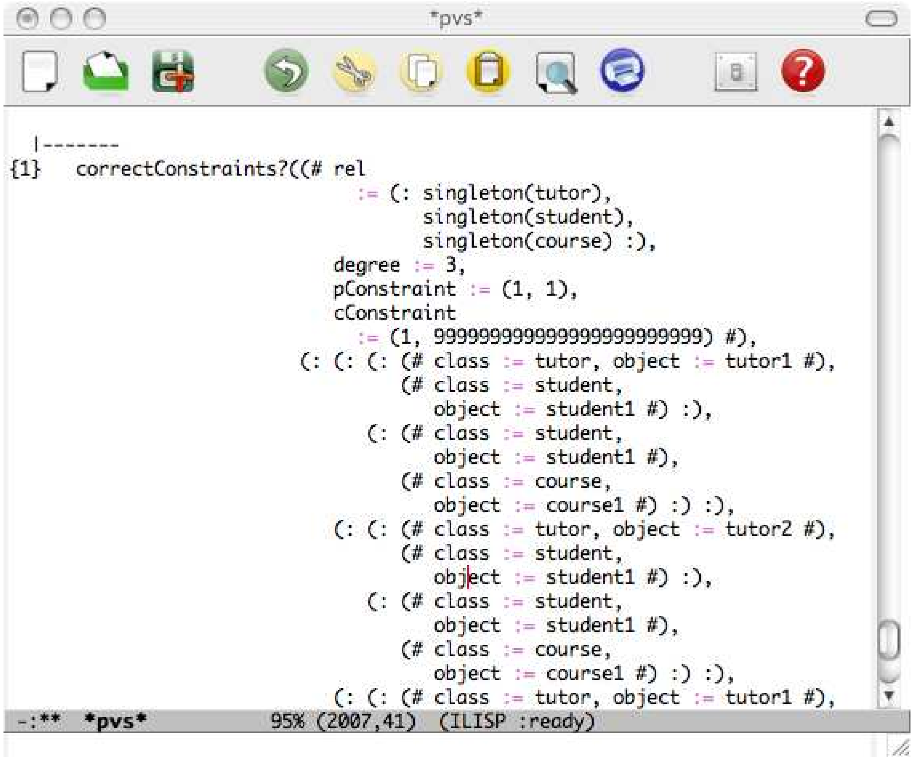

Figure 7: Validating the participation constraint of relationship instance *cst*

The type checking of '*dataEx*' and '*schemaNData*' theories proves all Type Correctness Conditions (TCCs) automatically because there are only a small number of type constraints applied. The theorem proving of '*correctSchemaNData_con*' conjecture in '*schemaNData*' theory results in an improvable state as shown in Figure 7. It shows that the relationship instances of '*cst*' relationship does not conform the participation constraint of the '*cst*' relationship. In the proof step for this part of the conjecture, it shows that two instances of tutor, '*tutor1*' and '*tutor2*', are related to a single '*cs*' relationship for '*student_details1*' in '*course1*' where the parent participation constraint is 1:1.

After correcting the above error, theorem proving of the '*correctSchemaNData_con*' conjecture in '*schemaNData*' theory results in an improvable state similar to the previous state. It shows that the relationship instances of '*sh*' relationship do not conform to the participation constraint of the '*sh*' relationship which is a disjunctive relationship. The disjunctive relationship always has 1:1 for its parent participation constraint since only one of the disjunctive objects can

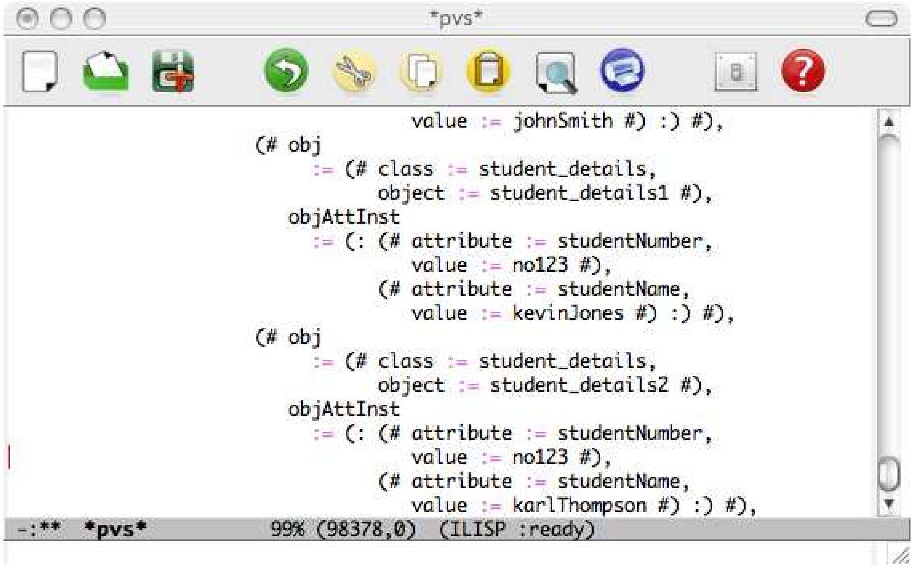

Figure 8: Validating the uniqueness of primary key *studentNumber*

be related to its parent. But the proof step for this part of the conjecture shows that the '*student_details1*' is related to both '*home1*' and '*hostel1*' violating the parent participation constraints for disjunctive relationship '*sh*'.

After correcting the above incorrect representation, theorem proving of the '*correctSchemaNData_con*' conjecture in the '*schemaNData*' theory has resulted in another improvable state as shown in Figure 8. It shows that the candidate key instances of '*student_details*' object does not satisfy the constraints. The candidate key has a constraint that specifies that the value of a candidate key must be unique. But the '*oAtt14cKey*' function used by '*correctCkey*' function to check this constraint shows that the candidate key value '*no123*' is related to two different students.

When all three introduced error are corrected and the '*correctSchemaNData_con*' conjecture is verified, the XML data example in Figure 3 is verified to be valid against its schema. With automated verification support, predefined theorems and lemmas the definitions of schema instance can be verified and its data instance can be validated with little effort.

5 Related Work

There has been other research that provides a formal semantics for semistructured data. For example, the formalization of DTD (Document Type Definition) and XML declarative description documents using a description logic has been presented by Calvanese et al. [Calvanese et al., 1999]. Anutariya et al. presented a similar formalization with a theoretical framework developed using declarative description theory [Anutariya et al., 2000]. Also spatial tree logics have been used to formalize semistructured data by Conforti and Ghelli [Conforti and Ghelli, 2003]. More recently, hybrid multimodal logic was used to formalize semistructured data by Bidoit et al. [Bidoit et al., 2004]. Moreover, we applied a similar approach to formalize semistructured data using ORA-SS and Z/EVEs [Lee et al., 2005]. While this work has helped us develop a better understanding of the semantics of semistructured data, none of it has provided automated verification. In other research we presented a formalization of the ORA-SS notation in Alloy [Wang et al., 2006]. Although automated verification was available using the Alloy Analyzer, it had a scalability problem making verification impossible for a large set of semistructured data. The Alloy Analyzer uses a model checking approach to perform verification where it checks the possibility of counterexamples from every possible example of a model derived from the given set of data [Jackson, 2006]. With a larger set of data, this approach requires the checking of a huge number of examples which can be beyond the capability of the Alloy Analyzer causing a scalability problem. The formalization of ORA-SS in OWL [Li et al., 2006] was presented with improved verification performance in terms of scalability and time taken. The PVS approach to formalize ORA-SS [Lee et al., 2006] was also presented with type checking and theorem proving abilities and does not exhibit any scalability issues. According to the comparisons presented in [Dobbie et al., 2006; Dobbie et al., 2007], the OWL approach has limitations in verifying the scope of properties of ORA-SS whereas PVS can verify the correctness of all the properties.

In other research involving schema transformation and verification, Arenas and Libkin established a normalization algorithm for XML using DTD as its schema [Arenas and Libkin, 2004]. Later, Arenas et al. derived consistency checking for XML specifications using DTD [Arenas et al., 2002]. Also Jagadish et al. developed an XML query optimizing algebra called TAX for manipulating XML data [Jagadish et al., 2001]. However, again none of this research has gone as far as automated verification and its applications has been restricted to a single form of semistructured data, i.e., XML. Furthermore, the approaches fail to consider the more complex semantics of semistructured data, as DTD was used to model the schema. Hence, none of the related work can be adequately used to automatically verify whether a set of transformations on a schema for semistructured data produces a result that is semantically consistent with the original schema.

6 Conclusion and Future Work

Semistructured data is a fast growing and widely used standard for information storage and exchange. The developed database systems for semistructured data use various algorithms to perform schema and data transformations such as normalization and view creation. It is critical to ensure the consistency of the data during manipulation. Thus, a mathematical foundation for semistructured data design is essential. One of the most effective approaches for such verification is to use a formalism to specify the semantics for semistructured data and perform automated verification on schema models, data instances and transformation algorithms. This paper presents the first step towards this goal. In our research, we have formally specified the semantics of ORA-SS in PVS and provided verification. The specification of the ORA-SS semantics is divided into specifications for schema, data, and mappings between schema and data, where verification of each definition against the semantics of ORA-SS and validation of data against its schema are established. The specification and verification for the semantics of ORA-SS in PVS is more powerful compared to existing research that use XML schema or DTD to validate the semistructured data, because the ORA-SS data modeling language is richer than the other data modeling languages for semistructured data enabling deeper semantic checking. For example, validation of semistructured data with XML Schema or DTD will not be able to recognize data inconsistency or redundancies in ternary or n-ary relationships, disjunctive relationships, nor relationship attributes, where these are possible in our approach. In addition, verification support of PVS allows the automated verification on large models for most parts of the specification, where interactive verification requires a little effort. In this research, it is clear that the formal specifications of ORA-SS semantics provides deeper semantic-based checking for semistructured data. Furthermore, our work shows that the semantics of ORA-SS is appropriate for expressing normalization algorithms. Therefore, this research also provides a solid basis for establishing a formal specification and verification of schema and data transformations to verify the correctness of semistructured data normalization.

In the future, we plan to extend and concentrate our work on defining criteria and standard transformation operators that are used to transform ORA-SS schemas for normalization. These standard operators will be defined and verified on top of the formal specification of the ORA-SS semantics. With the formally specified standard transformation operators, it will be possible to verify the result of the application of the operators. Furthermore, various transformation algorithms can be compared to find the best algorithms for specific contexts. In addition, we also plan to define and verify the extended PVS semantics of the ORA-SS language to model the normalization process in semistructured data design.

References

[Anutariya et al., 2000] Anutariya, C., Wuwongse, V., Nantajeewarawat, E., and Akama, K. (2000). Towards a foundation for XML document databases. In *EC-Web*, pages 324–333.

[Arenas et al., 2002] Arenas, M., Fan, W., and Libkin, L. (2002). On verifying consistency of XML specifications. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 259–270, New York, NY, USA. ACM Press.

[Arenas and Libkin, 2004] Arenas, M. and Libkin, L. (2004). A normal form for XML documents. *ACM Transactions on Database Systems*, 29:195–232.

[Bidoit et al., 2004] Bidoit, N., Cerrito, S., and Thion, V. (2004). A first step towards modeling semistructured data in hybrid multimodal logic. *Journal of Applied Non-Classical Logics*, 14(4):447–475.

[Calvanese et al., 1999] Calvanese, D., Giacomo, G. D., and Lenzerini, M. (1999). Representing and reasoning on XML documents: A description logic approach. *Journal of Logic and Computation*, 9(3):295–318.

[Conforti and Ghelli, 2003] Conforti, G. and Ghelli, G. (2003). Spatial tree logics to reason about semistructured data. In *SEBD*, pages 37–48.

[Dobbie et al., 2006] Dobbie, G., Sun, J., Li, Y. F., and Lee, S. U.-J. (2006). Research into Verifying Semistructured Data. In *ICDCIT '06: Proceedings of the 3rd International Conference on Distributed Computing and Internet Technology*, volume 4317 of *Lecture Notes in Computer Science*, pages 361–374, Berlin, Germany. Springer Verlag.

[Dobbie et al., 2007] Dobbie, G., Sun, J., Li, Y. F., and Lee, S. U.-J. (2007). Extended abstract: towards verifying semistructured data. In *APCCM '07: Proceedings of the 4th Asia-Pacific Conference on Comceptual Modelling*, pages 11–14, Darlinghurst, Australia. Australian Computer Society, Inc.

[Dobbie et al., 2001] Dobbie, G., Wu, X., Ling, T. W., and Lee, M. L. (2001). ORA-SS: Object-relationship-attribute model for semistructured data. Technical Report TR 21/00, School of Computing, National University of Singapore.

[Harold and Means, 2004] Harold, E. R. and Means, W. S. (2004). *XML in a Nutshell*. O'Reilly, Sebastopol, 3rd edition.

[Jackson, 2006] Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[Jagadish et al., 2001] Jagadish, H. V., Lakshmanan, L. V. S., Srivastava, D., and Thompson, K. (2001). TAX: A tree algebra for XML. In *DBPL*, pages 149–164.

[Lawford and Wu, 2000] Lawford, M. and Wu, H. (2000). Verification of real-time control software using PVS. In Ramadge, P. and Verdu, S., editors, *Proceedings of the 2000 Conference on Information Sciences and Systems*, volume 2, pages TP1–13–TP1–17, Princeton, NJ. Dept. of Electrical Engineering, Princeton University.

[Lee et al., 2006] Lee, S. U.-J., Dobbie, G., Sun, J., and Groves, L. (2006). A PVS Approach to Verifying ORA-SS Data Models. In *SEKE '06: Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering*, pages 126–131, Skokie, Illinois, USA. Knowledge Systems Institute Graduate School.

[Lee et al., 2008a] Lee, S. U.-J., Sun, J., Dobbie, G., and Groves, L. (2008a). Verifying Semistructured Data Normalization using PVS. In *ICECCS '08: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems*, Washington, DC, USA. IEEE Computer Society.

[Lee et al., 2008b] Lee, S. U.-J., Sun, J., Dobbie, G., Groves, L., and Li, Y. F. (2008b). Verification Criteria for Normalization of Semistructured Data. In *ASWEC '08: Proceedings of the 19th Australian Software Engineering Conference.*

[Lee et al., 2005] Lee, S. U.-J., Sun, J., Dobbie, G., and Li, Y. F. (2005). A Z Approach in Validating ORA-SS Data Models. In *SVV '05: Proceedings of the Third International Workshop on Software Verification and Validation*, volume 157, pages 95–109.

[Li et al., 2006] Li, Y. F., Sun, J., Dobbie, G., Sun, J., and Wang, H. H. (2006). Validating Semistructured Data Using OWL. In *WAIM '06: Proceedings of the 7th International Conference on Web-Age Information Management*, pages 520–531, Berlin, Germany. Springer-Verlag.

[Ling et al., 2001] Ling, T., Lee, M., and Dobbie, G. (2001). Applications of ORA-SS: An object-relationship-attribute data model for semistructured data. In *IIWAS '01: Proceedings of 3rd International Conference on Information Integration and Web-based Applications and Serives.*

[Ling et al., 2005] Ling, T. W., Lee, M. L., and Dobbie, G. (2005). *Semistructured Database Design.* Springer, New York.

[Owre et al., 1992] Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: A prototype verification system. In Kapur, D., editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY. Springer-Verlag.

[Srivas et al., 1997] Srivas, M., Rueß, H., and Cyrluk, D. (1997). Hardware verification using PVS. In Kropf, T., editor, *Formal Hardware Verification: Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, pages 156–205. Springer-Verlag.

[Thompson et al., 2007] Thompson, H. S., Sperberg-McQueen, C., Mendelsohn, N., Beech, D., and Maloney, M. (2007). XML schema 1.1 part 1: Structures. http://www.w3.org/TR/xmlschema11-1.

[Vitt and Hooman, 1996] Vitt, J. and Hooman, J. (1996). Assertional specification and verification using pvs of the steam boiler control system. In Abrial, J.-R., Boerger, E., and Langmaack, H., editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165, pages 453–472. Springer-Verlag.

[Wang et al., 2006] Wang, L., Dobbie, G., Sun, J., and Groves, L. (2006). Validating ORA-SS Data Models using Alloy. pages 231–242.

[Wu et al., 2001] Wu, X., Ling, T. W., Lee, M. L., and Dobbie, G. (2001). Designing semistructured databases using the ORA-SS model. In *WISE '01: Proceedings of 2nd International Conference on Web Information Systems Engineering*, Kyoto, Japan. IEEE Computer Society.