Research into Verifying Semistructured Data

Gillian Dobbie[1,⋆], Jing Sun[1], Yuan Fang Li[2,⋆⋆], and Scott UK-Jin Lee[1]

[1] Department of Computer Science, The University of Auckland, New Zealand
{gill, j.sun, scott}@cs.auckland.ac.nz
[2] School of Computing, National University of Singapore, Republic of Singapore
liyf@comp.nus.edu.sg

Abstract. Semistructured data is now widely used in both web applications and database systems. Much of the research into this area defines algorithms that transform the data and schema, such as data integration, change management, view definition, and data normalization. While some researchers have defined a formalism for the work they have undertaken, there is no widely accepted formalism that can be used for the comparison of algorithms within these areas. The requirements of a formalism that would be helpful in these situations are that it must capture all the necessary semantics required to model the algorithms, it should not be too complex and it should be easy to use. This paper describes a first step in defining such a formalism. We have modelled the semantics expressed in the ORA-SS (Object Relationship Attribute data model for SemiStructured data) data modelling notation in two formal languages that have automatic verification tools. We compare the two models and present the findings.

Keywords: data model, semistructured data, automatic verification.

1 Introduction

Semistructured data is now widely used in both web applications and database systems. There are many research challenges in this area, such as data integration, change management, view definition, and data normalization. Traditionally in these areas a formalism is defined for the database model, and properties of the algorithms can be reasoned about, such as the dependency preserving property of the normalization algorithm in the relational data model. Because research into semistructured data is still in its infancy, many algorithms have been defined in this area and a number of formalisms have been proposed but there is no widely accepted formalism that is generally accepted to reason about the properties of the algorithms. Such a formalism must capture all the necessary semantics required to model the algorithms, should not be too complex, and should be easy to use.

Another area that has been developing steadily is automatic verification. This involves formally specifying a model of a system, and running an automatic model checker or theorem prover that proves or disproves the consistency of the model. In this paper we describe research that we have undertaken in this direction. We have determined the

⋆ This work is funded by a Marsden Grant (UOA317) from the Royal Society of New Zealand.
⋆⋆ This work is funded by the Singapore Millenium Foundation (SMF).

S. Madria et al. (Eds.): ICDCIT 2006, LNCS 4317, pp. 361–374, 2006.

semantics that we believe are required to reason about the properties of algorithms, we have modelled them using two different logics and used automatic verification to reason about the properties. This work is a first step towards establishing a widely accepted formalism, and it does highlight some important findings:

- the importance of the model containing enough semantics to express the algorithms over the data, but not excessive semantics
- the model must be broken down into logical sections for understandability and extensibility
- the importance of basing research into semistructured data on previous research in the database area rather than reinventing the wheel.

More specifically, we use a data modelling notation that extends the entity relationship (ER) data model. The ORA-SS (Object Relationship Attribute data model for SemiStructured Data) data model models the schema and instance, so it is possible to model an XML Schema document in an ORA-SS schema diagram and an XML document in an ORA-SS instance diagram. However the semantics captured in an ORA-SS schema diagram are richer than those that are represented in XML Schema. We have modelled the semantics expressed in the ORA-SS diagrams in OWL (Web Ontology Language), which is based on a description logic, which itself can be translated into first order predicate logic. Because OWL was designed for sharing data using ontologies on the web, it was a natural starting point. OWL has an automatic reasoning tool called RACER (Renamed ABox and Concept Expression Reasoner). We also modelled the semantics expressed in the ORA-SS diagrams in PVS (Prototype Verification System), which is a typed higher order logic based verification system with a theorem prover. PVS can express more complex structures than OWL and in part, you get typing for free.

Section 2 clarifies the motivation for the project and describes other research that has offered a formal model for semistructured data. Section 3 provides background for the rest of the paper, highlighting the main features of ORA-SS, OWL and PVS. Section 4 summarizes the modelling of ORA-SS in OWL and PVS. More details of these models can be found in [1,2]. In Section 5 we analyze the two models. We conclude and provide future work in Section 6.

2 Related Work

Much of the work that has addressed challenges in the semistructured data area, such as [LiLi06, LeHo06, EmMo01], have proposed algorithms. However, there is little comparative analysis of algorithms that are designed for similar tasks. Because there is no widely accepted formalism, it is not possible to reason about the correctness, or show specific properties of the algorithms in a general way. Moreover it is difficult to compare properties of algorithms that are designed for similar purposes. For example it is difficult to compare the properties of the normalization algorithms that have been defined for semistructured data.

The area that our group is specifically interested in is normalization of semistructured data (or XML documents). There has been some very good and very practical work done

in this area, such as [3,4,5,6]. If there was a widely accepted data model, with a set of defined operations it would be possible to show properties such as, whether data is lost during the transformations specified and it would also be possible to reason about what constraints are lost in the transformations. The kinds of formal definitions that we have seen to date in the normalization area include [7,5,8,6].

Another area where there are similar transformations are view definitions. It would be helpful to be able to show that given a set of view transformations again no data is lost, and also show that particular operations are reversible. At one level, data integration can be thought of as creating a view over a set of schemas. The unified view that is formed can be defined by a set of operations and the unified view would be better understood if there were some way to state properties of the operations. The operations defined in change management, such as *insert, drop* and *move* can be defined formally. There should be operators for both changes to the data and changes to the schema. A formal definition would not only be useful when defining what changes to allow, but could also help in the defintion of operations and perhaps in detecting the kinds of changes that have occured between different versions of the data or schema.

There was an interesting workshop that highlights the need to bring together people who are working in foundations of semistructured data from different areas that are related to semistructured data [9]. As you can see there have been a number of different approaches to defining foundations for semistructured data (e.g. [10,11], where most model the schema and data as a tree or graph and they are unable to model some of the constraints that can be specified in schema languages such as cardinality of children in relationships in the schema. These works consider limited semantics and do not provide automatic verification.

3 Background

The three key components in this work is the data modelling notation, ORA-SS, the ontology language OWL with the automatic reasoner RACER, and the formal verification language PVS. We briefly describe each of these components in this section.

3.1 ORA-SS (Object Relationship Attribute Model for Semistructured Data)

ORA-SS provides a notation for representing constraints on schemas and instances of semistructured data. The ORA-SS schema diagram extends ER diagrams with hierarchical parent-child relationships, and ordering. Schema diagrams represent relationship types between object classes, cardinality of relationship types, and distinguishes between attributes of object classes and attributes of relationship types. The ORA-SS instance diagram represents the same information as DOM diagrams, namely the relationships between objects and values of attributes. Objects match elements in XML documents, while attributes match leaf elements or attributes. A full description of the ORA-SS data modeling language can be found in [12,13].

We will now highlight some of the salient points in the ORA-SS schema diagram in Figure 1. There is a relationship type between object class *course* and object class *student*. It is a binary relationship type with name *cs*. Each course can have 4 to many

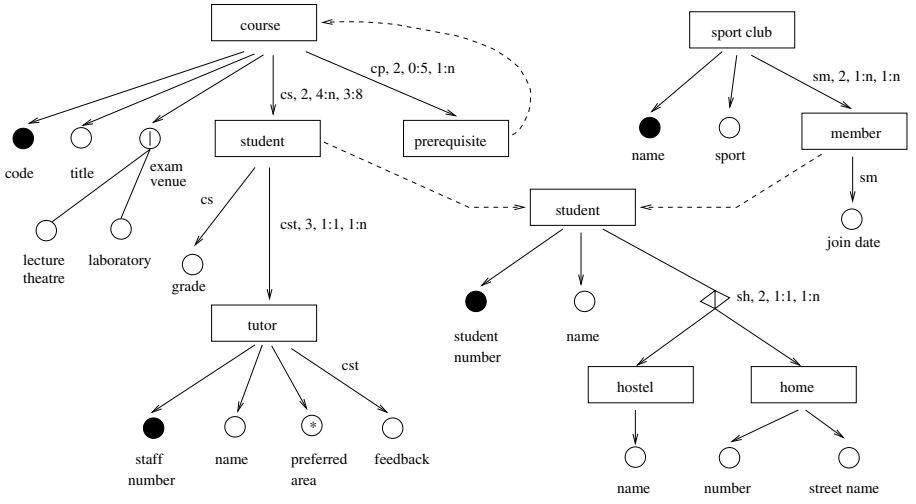

Fig. 1. The ORA-SS schema diagram of a *Course-Student* data model

students and a student can take 3 to 8 courses. Attribute *code* is an identifying attribute of course, and there is a reference between *prerequisite* and *course*, i.e. each prerequisite is in fact another course. The attribute grade belongs to the relationship type *cs*, i.e. it is the grade of a student in a course. Notice that relationship *cst* is a ternary relationship type, i.e. it relates object classes *course*, *student* and *tutor*.

3.2 OWL (Web Ontology Language)

OWL was designed to share data using ontologies over the web, and represents the meanings of terms in vocabularies and the relationships between those terms in a way that is suitable for processing by software.

Description logics [14] are logical formalisms for representing information about knowledge in a particular domain. It is a subset of first-order predicate logic and is well-known for the trade-off between expressivity and decidability. Based on RDF Schema [15] and DAML+OIL [16], the Web Ontology Language (OWL) [17] is the de facto ontology language for the Semantic Web. It consists of three increasingly expressive sub-languages: OWL Lite, DL and Full. OWL DL is very expressive yet decidable. As a result, core inference problems, namely concept subsumption, consistency and instantiation, can be performed automatically. RACER is an automatic reasoning tool for OWL ontologies, which supports min/max restrictions on integers, roles, role hierarchies, inverse and transitive roles.

3.3 PVS (Prototype Verification System)

PVS is a typed higher-order logic based verification system where a formal specification language is integrated with support tools and a theorem prover [18]. 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. With the language constructs provided, PVS specifications are represented in parameterized theories that contain assumptions, definitions, axioms, and theorems. Many applications have adopted PVS to provide formal verification support to their system properties [19,20,21].

4 Modelling ORA-SS Diagrams

The ORA-SS notation separates concerns naturally, separating the schema and the instance into individual diagrams. In our modelling we go a step further. We distinguish between:

- constraints that must hold on all schema diagrams
- constraints that must hold on all instance diagrams
- constraints that hold on the relationships between schemas and instances

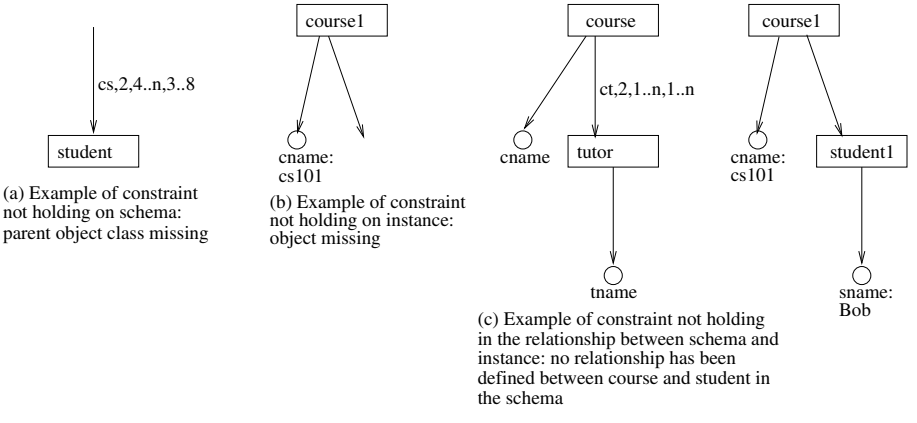

Fig. 2. Examples of constraints not holding in ORA-SS diagrams

The kind of constraint that must hold on the schema is that a child object class must be either related to a parent object class to form a binary relationship or related to another subrelationship type to form an n-ary relationship. The schema in Figure 2(a) violates this constraint. An example of a constraint that must hold on all instance diagrams is that a relationship is between 2 or more objects. Figure 2(b) shows a relationship in an instance diagram, where there is no child object. An example of a constraint that must hold on the relationship between schemas and instances is that if there is a relationship between two objects in an instance, then there must be a relationship type between the related object classes at the schema level. Figure 2(c) shows an instance with a relationship between objects *course1* and *student1*. Assuming that *student1* is a student and not a tutor, then this relationship violates the relationship type in the corresponding schema diagram. Although the mistakes appear obvious in the simple examples in Figure 2, they

are harder to see in more complex diagrams or when the data is in a different format, such as XML and XMLSchema.

For each of the models described above, we have a corresponding instance model. There are models for:

- the instance of the schema
- the instance of the instance
- the relationship between the instances of the schema and the instance

In the instance of the schema, it is possible to state constraints such as *course* is an object class. In the instance of the instance, it is possible to state that *course1* is an object, and in the relationship between instances of the schema and the instance it is possible to state that *course1* is a *course*. Figure 3 summarizes the components that make up the ORA-SS model.

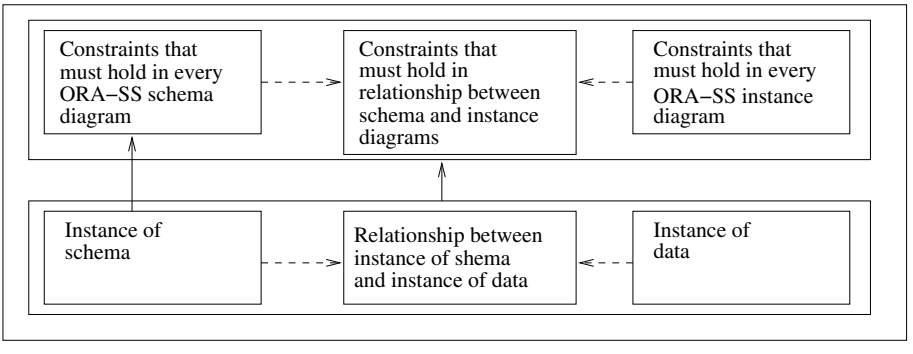

Fig. 3. Components of the Model

4.1 The ORA-SS Ontology in OWL

In OWL, we refer to a model as an ontology. The ORA-SS ontology contains the OWL definitions of all the ORA-SS concepts, such as object class, relationship type, and attributes. The 6 models defined above are captured in the ORA-SS Ontology.

Constraints on the Schema. As each object class and relationship type can be associated with attributes and other object classes or relationship types, we define an OWL class *ENTITY* to represent the super class of both object class and relationship type. The OWL class structure is shown as follows.

$$ENTITY \sqsubseteq \top \qquad ATTRIBUTE \sqsubseteq \top$$
$$OBJECT \sqsubseteq ENTITY \qquad ENTITY \sqcap ATTRIBUTE = \bot$$
$$RELATIONSHIP \sqsubseteq ENTITY \qquad OBJECT \sqcap RELATIONSHIP = \bot$$

It may not seem very intuitive to define relationship types as OWL classes. In ORA-SS, relationship types are used to relate various object classes and relationship types, it might seem more natural to model relationship types as OWL properties. However,

there are two reasons that we decide to model relationship types as OWL classes. Firstly, the domain of ORA-SS relationship types can be relationship types themselves, when describing the relationships of ternary and more. Secondly, classes and properties in OWL DL are disjoint. In our model, an OWL relationship type consists of instances which are actually pointers to the pairs of object classes or relationship types that this relationship relates.

In ORA-SS, object classes and relationship types are inter-related to form new relationship types. As mentioned above, since we model relationship types as OWL classes, we need additional properties to connect various object classes and relationship types.

Firstly, this is accomplished by introducing two object-properties, *parent* and *child*, which map a *RELATIONSHIP* to its domain and range *ENTITY*s. The following statements define the domain and range of *parent* and *child*. As in ORA-SS, the domain of a relationship (*parent*) can be either an object class or another relationship type, i.e., an *ENTITY*. The range (*child*) must be an *OBJECT*. These two properties are functional as one relationship type has exactly one domain and one range node. Moreover, we assert that only relationship types can have parents and child but object classes cannot.

$$\geq 1\ parent \sqsubseteq RELATIONSHIP \qquad \geq 1\ child \sqsubseteq RELATIONSHIP$$
$$\top \sqsubseteq \forall\, parent.ENTITY \qquad \top \sqsubseteq \forall\, child.OBJECT$$
$$\top \sqsubseteq \leq 1\ parent \qquad \top \sqsubseteq \leq 1\ child$$
$$OBJECT \sqsubseteq \neg\ \exists\, parent.\top \qquad RELATIONSHIP \sqsubseteq \forall\, parent.ENTITY$$
$$OBJECT \sqsubseteq \neg\ \exists\, child.\top \qquad RELATIONSHIP \sqsubseteq \forall\, child.OBJECT$$

Secondly, we define two more object-properties: *p-ENTITY-OBJECT* and *p-OBJECT-ENTITY*. These two properties are the inverse of each other and they serve as the super properties of the properties that are to be defined in later ontologies of ORA-SS schema diagrams. Those properties will model the restrictions imposed on the relationship types.

The domain and range of *p-ENTITY-OBJECT* are *ENTITY* and *OBJECT*, respectively. Since the two properties are inverses, the domain and range of *p-OBJECT-ENTITY* can be deduced.

$$p\text{-}OBJECT\text{-}ENTITY = (^{-}p\text{-}ENTITY\text{-}OBJECT)$$
$$\geq 1\ p\text{-}ENTITY\text{-}OBJECT \sqsubseteq ENTITY \qquad \geq 1\ p\text{-}OBJECT\text{-}ENTITY \sqsubseteq OBJECT$$
$$\top \sqsubseteq \forall\, p\text{-}ENTITY\text{-}OBJECT.OBJECT \qquad \top \sqsubseteq \forall\, p\text{-}OBJECT\text{-}ENTITY.ENTITY$$
$$ENTITY \sqsubseteq \forall\, p\text{-}ENTITY\text{-}OBJECT.OBJECT \qquad OBJECT \sqsubseteq \forall\, p\text{-}OBJECT\text{-}ENTITY.ENTITY$$

To define that attributes belong to entities or relationships, first of all, we define an object-property *has-ATTRIBUTE*, whose domain is *ENTITY* and range is *ATTRIBUTE*. Every *ENTITY* must have *ATTRIBUTE* as the range of *has-ATTRIBUTE*.

$$\geq 1\ has\text{-}ATTRIBUTE \sqsubseteq ENTITY \qquad ENTITY \sqsubseteq \forall\, has\text{-}ATTRIBUTE.ATTRIBUTE$$
$$\top \sqsubseteq \forall\ .has\text{-}ATTRIBUTE.ATTRIBUTE$$

For modeling the ORA-SS candidate and primary keys, we define two new object properties that are sub-properties of *has-ATTRIBUTE*. We also make the property *has-primary-key* inverse functional and state that each *ENTITY* must have at most one primary key. Moreover, we restrict the range of *has-candidate-key* to be *ATTRIBUTE*.

$$has\text{-}candidate\text{-}key \sqsubseteq has\text{-}ATTRIBUTE \qquad has\text{-}primary\text{-}key \sqsubseteq has\text{-}candidate\text{-}key$$
$$\top \sqsubseteq \forall\, has\text{-}candidate\text{-}key.ATTRIBUTE \qquad \top \sqsubseteq \leq 1\ has\text{-}primary\text{-}key^{-}$$
$$ENTITY \sqsubseteq \leq 1\ has\text{-}primary\text{-}key$$

Instance of the Schema. The instance of object classes are represented as a subclass of *OBJECT*.

$course \sqsubseteq OBJECT$ $\quad$ $tutor \sqsubseteq OBJECT$
$student \sqsubseteq OBJECT$ $\quad$ $sport\_club \sqsubseteq OBJECT$
$hostel \sqsubseteq OBJECT$ $\quad$ $home \sqsubseteq OBJECT$
$\ldots$ $\quad$ $\ldots$

The instance of relationship types are represented as a subclass of *RELATIONSHIP*.

$cs \sqsubseteq RELATIONSHIP$ $\quad$ $sm \sqsubseteq RELATIONSHIP$ $\quad$ $cst \sqsubseteq RELATIONSHIP$
$sh \sqsubseteq RELATIONSHIP$ $\quad$ $cp \sqsubseteq RELATIONSHIP$

The relationship type *cs* is bound by the *parent*/*child* properties as follows. We use both *allValuesFrom* and *someValuesFrom* restriction to make sure that only the intended class can be the parent/child class of *cs*.

$cs \sqsubseteq \forall\, parent.course$ $\quad$ $cs \sqsubseteq \forall\, child.student\_1$
$cs \sqsubseteq \exists\, parent.course$ $\quad$ $cs \sqsubseteq \exists\, child.student\_1$

As discussed in the previous subsection, for each ORA-SS relationship type we define two object-properties that are the inverse of each other.

Example 1. Take *cs* as an example, we construct two object-properties: *p-course-student* and *p-student-course*. Their domain and range are also defined.

$p\text{-}student\text{-}course = (^{-}p\text{-}course\text{-}student)$
$p\text{-}course\text{-}student \sqsubseteq p\text{-}ENTITY\text{-}OBJECT$
$p\text{-}student\text{-}course \sqsubseteq p\text{-}OBJECT\text{-}ENTITY$
$\geq 1\, p\text{-}course\text{-}student \sqsubseteq course$ $\quad$ $\geq 1\, p\text{-}student\text{-}course \sqsubseteq student\_1$
$\top \sqsubseteq \forall\, p\text{-}course\text{-}student.student\_1$ $\quad$ $\top \sqsubseteq \forall\, p\text{-}student\text{-}course.course$

One of the important advantages that ORA-SS has over XML Schema language is the ability to express participation constraints for parent/child nodes of a relationship type. This ability expresses the cardinality restrictions that must be satisfied by ORA-SS instances.

Using the terminology defined previously, ORA-SS parent participation constraints are expressed using cardinality restrictions in OWL on a sub-property of *p-ENTITY-OBJECT* to restrict the parent class *Prt*. Child participation constraints can be similarly modeled, using a sub property of *p-OBJECT-ENTITY*.

Example 2. In Fig. 1, the constraints captured by the relationship type `cs` state that a `course` must have at least 4 students; and a `student` must take at least 3 and at most 8 courses. The following axioms are added to the ontology. The two object-properties defined above capture the relationship type between `course` and `student`.

$course \sqsubseteq$ $\quad$ $student\_1 \sqsubseteq \forall\, p\text{-}student\text{-}course.course$
$\forall\, p\text{-}course\text{-}student.student\_1$ $\quad$ $student\_1 \sqsubseteq \geq 3\, p\text{-}student\text{-}course$
$course \sqsubseteq \geq 4\, p\text{-}course\text{-}student$ $\quad$ $student\_1 \sqsubseteq \leq 8\, p\text{-}student\text{-}course$

The instances of attributes are modelled as a subclass of *ATTRIBUTE*. As OWL adopts the Open World Assumption [17] and an ORA-SS model is closed, we need to find ways to make the OWL model capture the intended meaning of the original diagram. The following are some modeling *conventions*.

– For each *ENTITY*, we use an *allValuesFrom* restriction on *has-ATTRIBUTE* over the union of all its *ATTRIBUTE* classes. This denotes the complete set of attributes that the *ENTITY* holds.

  Example 3. In the running example, the object class *student* has student number and name as its attributes.

  $$student \sqsubseteq \forall\, has\text{-}ATTRIBUTE.(student\_number \sqcup name)$$

– Each entity (object class or relationship type) can have a number of attributes. For each of the entity-attribute pairs in an ORA-SS schema diagram, we define an object-property, whose domain is the entity and range is the attribute. For an entity *Ent* and its attribute *Att*, we have the following definitions.

  $$has\text{-}Ent\text{-}Att \sqsubseteq has\text{-}ATTRIBUTE \qquad \top \sqsubseteq \forall\, has\text{-}Ent\text{-}Att.Att$$
  $$\geq 1\ has\text{-}Ent\text{-}Att \sqsubseteq Ent$$

  Example 4. In Fig. 1, the object class *sport club* has an attribute name. It can be modeled as follows.

  $$\geq 1\ has\text{-}sport\_club\text{-}name \sqsubseteq sport\_club \qquad has\text{-}sport\_club\text{-}name \sqsubseteq has\text{-}ATTRIBUTE$$
  $$\top \sqsubseteq \forall\, has\text{-}sport\_club\text{-}name.name \qquad sport\_club \sqsubseteq \forall\, has\text{-}sport\_club\text{-}name.name$$

For an entity with a primary key attribute, we use an *allValuesFrom* restriction on the property *has-primary-key* to constrain it. Since we have specified that *has-primary-key* is inverse functional, this suffices to show that two different objects will have different primary keys. Moreover, for every attribute that is the primary key attribute, we assert that the corresponding object property is a sub property of *has-primary-key*.

Example 5. In Fig. 1, object class *course* has an attribute *code* as its primary key and this is modeled as follows. The `hasValuesFrom` restriction enforces that each individual must have some *code* value as its primary key.

$$course \sqsubseteq \forall\, has\text{-}primary\text{-}key.code \qquad course \sqsubseteq \exists\, has\text{-}primary\text{-}key.code$$

4.2 The ORA-SS Model in PVS

The PVS model is divided into six files, as shown in Figure 3. We did this because by separating the concerns, it is easier to maintain and modify the model.

Constraints on the Schema. A relationship type is defined as a list of a set of object classes, where there is more than one object class and there are no cycles in the list. That is a relationship type cannot contain the same object class more than once. If there are more than 2 object classes in a relationship type, then the child object class relates to another relationship type.

```
no_cycle_oc(loc: list[set[OC]]): RECURSIVE bool =
    CASES loc OF
       null: TRUE,
       cons(ocs, subloc): (FORALL(subocs: set[OC]):
           member(subocs, subloc) =>disjoint?(ocs, subocs)) AND
           no_cycle_oc(subloc)
     ENDCASES
```

```
MEASURE length(loc)

RelType: TYPE = ocsList: list[set[OC]] | (length(ocsList) > 1)
                AND (no_cycle_oc(ocsList))

Relationship: TYPE = rel: RelType | (length(rel) > 2) =>
                  (EXISTS(subRel: RelType): subRel = cdr(rel))
```

In an ORA-SS schema diagram, there are two types of relationship, i.e., a normal relationship where the child participant is a single object class; and a disjunctive relationship where the child participant is a set of disjunctive object classes. The above definition includes both cases. The '*no_cycle_oc*' function is defined as a recursive predicate function to disallow repetition of object classes in a relationship. The relationship type states that for any relationship with more than two elements the tail of the list forms another sub-relationship type.

Every relationship type in an ORA-SS schema diagram has its associated constraints on its participating objects shown using the `min:max` notation. It constrains the number of child objects that a parent object can relate to and vice versa.

```
parentConstraints(rel: Relationship): [nat, posnat]

parentSet(loRel: list[list[OBJ]], loParent: list[OBJ]):
    RECURSIVE nat =
   CASES loRel OF
      null: 0,
      cons(oRelHead, oRelRest):
        (IF (NOT(oRelHead = null) AND NOT(loParent = null) AND
            loEqual?(cdr(oRelHead), cdr(loParent))) THEN 1
         ELSE 0 ENDIF)
        + parentSet(oRelRest, loParent)
   ENDCASES
MEASURE length(loRel)

correctPC?(rel: Relationship): bool =
   FORALL(oRel: list[ObjRelationship]):
   member(oRel, relInstance(rel)) IMPLIES
     (proj_1(parentConstraints(rel)) <=
       parentSet(nArrayObjRelAll(relInstance(rel)),
       nArrayObjRel(oRel))) AND
     (proj_2(parentConstraints(rel)) >=
       parentSet(nArrayObjRelAll(relInstance(rel)),
       nArrayObjRel(oRel)))
```

The above defines parent constraints as a function where it takes a relationship type as an argument and returns the tuple of natural number and positive natural number which refers to a `min:max` pair. There is also a function *correctPC?* that checks whether the number of relationship instances for each object of the parent object class

or each relationship instance of the sub-relationship type is within the boundaries defined in the relationship or not. The child constraints of the relationship and constraints on attribute values associated with objects and relationships can be defined in a similar way.

An object can have an attribute or set of attributes that have a unique value for each instance of an object class called a candidate key.

```
candidateKeys(oc: OC): list[list[ATT]]

correctCKey?(oc: OC): bool =
  (FORALL(attList: list[ATT]):
   member(attList, candidateKeys(oc)) =>
      noAttRepeat?(attList) AND
      isCKeyObjAtt?(attList, objAttribute(oc))) AND
   noKeyRepeat?(candidateKeys(oc))
```

The above defines the object having a candidate key as a function where it takes an object class as an argument and returns all candidate keys of *OC* as a list of list of attribute. The list of attributes is able to model both candidate key and composite candidate key. The function for checking candidate keys checks two conditions. It checks whether two objects are different when values of the candidate key for each object are different, where the values of the candidate keys belong to the set of attribute values of the object attributes. The function checks the uniqueness of the keys as well as whether the key attributes are actually attributes of the object class. 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 as a function that takes an object class as an argument and returns a list of attributes which is the primary key of that object class.

Constraints on the Instance. The instances of an object class in a declaration.

```
OC: DATATYPE
  BEGIN
    department: department?
    course: course?
    student: student?
    tutor: tutor?
    home: home?
    hostel: hostel?
  END OC
```

The instances of a relationship are represented as variables (or instances) of relationship types, the degree is represented as a conjecture, and participation constraints are represented as an axiom.

```
dc: Relationship = (:singleton(course),singleton(department):)
dcDegree: CONJECTURE Degree(dc) = 2
dcConstraint_Ax: AXIOM
 parentConstraints(dc)=(1, many) AND childConstraints(dc)=(1, 1)
```

Instances of attributes are modelled in a similar way to instances of object classes. Attributes are assigned to object classes or relationship types in axioms.

```
objAtts: AXIOM
   objAttribute(department) = (:deptName:) AND
   objAttribute(course) = (:code, title, examVenue:)

relAtts: AXIOM
   relAttribute(cst) = (:feedback:) AND
   relAttribute(cs) = (:grade:)
```

There is another axiom that states the primary keys of the object classes.

```
pKeys: AXIOM
   primaryKey(department) = (:deptName:) AND
   primaryKey(course) = (:code:)
```

5 Discussion

In this section we discuss the findings from modeling ORA-SS language in OWL and PVS. OWL is based on Description Logic (DL). An OWL ontology model describes the relationships and constraints among classes. In this sense, it is very similar to that of an ORA-SS schema diagram. OWL provides qualifying number restrictions, role hierarchies, inverse roles and transitive roles. We use qualifying number restrictions for defining the cardinality of relationships, role hierarchies for expressing inheritance of features, and inverse roles to express relationships. These concepts map naturally to some of the features in ORA-SS diagrams. Furthermore, the OWL ontology is designed for creating individuals from a relational model. Thus there is no need to explicitly define the ORA-SS instances in our OWL representation, but to use the OWL individuals directly. Therefore the OWL semantics of the ORA-SS language is simpler.

However, DL has its own limitations. It can be clumsy to model complicated types and predicates. For example, the current OWL notation has no cardinality constraint on the domain, so we had to introduce inverse to check the domain cardinality restriction. On the other hand, PVS is based on high-order logic, which has more expressiveness than that of DL. Therefore, our PVS representation is capable of modeling complicated constraints among schemas such as transformation operators, which may not be trivial using OWL. In addition, PVS is a strongly typed language. It has a type checking facility that can be used to verify typing conditions such as that objects belong to object classes and so on. This means that the specifications are easy to follow because we do not have to write extra theorems for typing. This is very handy because it does syntax checking on the ORA-SS language.

In terms of the automated verification, RACER is a reasoning engine and PVS has a theorem prover. A reasoning engine derives a conclusion from premises expressed in the knowledge base. It behaves more like a model checker, which verifies the consistency of OWL individuals within a finite scope against its ontology model. Because the OWL reasoner is design to handle huge ontology instances, it is very scalable enabling the checking of large XML files. On the other hand, a theorem prover proves mathematical theorems from a specification. It is good at performing a complete proof on certain properties of a model without any scope limitations. Thus PVS is a heavy duty theorem prover that can check more deep constraints such as schema transformation

verifications. It provides tools that allow the definition of higher-level proof strategies enabling incremental building and reuse of proofs.

From our own experience, we found that the OWL and PVS approaches are actually complementary to each other. The RACER reasoner is good at detecting inconsistencies (within a finite scope) in an ORA-SS model. If an inconsistency exists it can also provides counter-examples. This is very useful in the sense of debugging errors. On the other hand, PVS theorem prover is good at verifying the total correctness of a property within the model without scope limitations. If a property is proved from PVS that means it should hold for every instance of the model. Hence, by using the RACER and PVS together, we could achieve a complete verification on the ORA-SS models.

6 Conclusion

Much of the research in the area of semistructured data involves algorithms that transform the data and schema, such as data integration, change management, view definition, and data normalization. The work presented in this paper is a first step towards formally defining a data model that is rich enough to capture the semantics that are needed to model and reason about the properties of operations that are capable of describing the algorithms described above. We have modelled the ORA-SS data model using OWL and PVS, and conclude that:

- Semantic Web languages, such as OWL, are extremely good at capturing semantic information about semistructured data.
- Reasoning tools, such as Racer, can be used to check the consistency of the ORA-SS schema and instance diagrams.
- PVS can be used to define a formal mathematical semantics for semistructured data.
- The automated verification provided by PVS empowers our definition of ORA-SS semantics, and if designed well it can be easy to extend.

There are a number of directions that we wish to take this work. Firstly we will study the algorithms that have been defined for the normalization of semistructured data, and derive the basic operations that are necessary to describe the algorithms. Secondly we will extend the OWL and PVS models of ORA-SS diagrams with a formal definition of the basic operations to compare how each model of ORA-SS performs. Thirdly, we will test the success of the models by modelling at least two of the normalization algorithms and comparing the properties of each. Finally we will investigate if the operators that are defined for normalization are general enough to express the general problem of view definitions.

References

1. Li, Y.F., Sun, J., Dobbie, G., Sun, J., Wang, H.H.: Validating Semistructured Data Using OWL. In: Proceedings of the 7th International Conference on Web Age Information Management, LNCS 4016 (2006)
2. Lee, S., Dobbie, G., Sun, J.: PVS Approach to Verifying ORA-SS Data Models. In: Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering. (2006)

3. Embley, D.W., Mok, W.Y.: Developing XML Documents with Guaranteed "Good" Properties. In: Proceedings of the 20th International Conference on Conceptual Modeling, LNCS 2224 (2001)
4. Wu, X., Ling, T.W., Lee, M.L., Dobbie, G.: 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 (2001)
5. Arenas, M., Libkin, L.: A Normal Form for XML Documents. ACM Transactions on Database Systems **29**(1) (2004) 195–232
6. Wang, J., Topor, R.: Removing XML Data Redundancies Using Functional and Equality Generating Dependencies. In: Proceedings of the Australasian Database Conference (2005)
7. Mani, M., Lee, D., Muntz, R.R.: Semantic Data Modeling Using XML Schemas. In: Proceedings of the 20th International Conference on Conceptual Modeling, LNCS 2224 (2001)
8. Vincent, M.W., Liu, J., Liu, C.: Strong Functional Dependencies and their Application to Normal Forms in XML. ACM Transactions on Database Systems **29**(3) (2004) 445–462
9. Neven, F., Schwentick, T., Suciu, D., eds. In Neven, F., Schwentick, T., Suciu, D., eds.: Foundations of Semistructured Data. Volume 05061., Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2005)
10. Milo, T., Suciu, D., Vianu, V.: Typechecking for XML Transformers. J. Comput. Syst. Sci. **66**(1) (2003) 66–97
11. Jagadish, H.V., Lakshmanan, L.V.S., Srivastava, D., Thompson, K.: Tax: A tree algebra for xml. In: DBPL. (2001) 149–164
12. Dobbie, G., Wu, X., Ling, T., Lee, M.: ORA-SS: Object-Relationship-Attribute Model for Semistructured Data. Technical Report TR 21/00, School of Computing, National University of Singapore (2001)
13. Ling, T.W., Lee, M.L., Dobbie, G.: Semistructured Database Design. Springer-Verlag (2005)
14. Nardi, D., Brachman, R.: An introduction to description logic. In Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P., eds.: The description logic handbook: theory, implementation, and applications. Cambridge University Press (2003) 1–40
15. Brickley, D., Guha, R., (eds.): Resource description framework (rdf) schema specification 1.0 (2004) http://www.w3.org/TR/rdf-schema/.
16. Connolly, D., van Harmelen, F., Horrocks, I., McGuiness, D., Patel-Schneider, P., Stein, L., (eds): Reference description of the DAML+OIL ontology markup language (2001) http://www.w3.org/TR/daml+oil-reference.
17. Horrocks I., Patel-Schneider P.F., v.H.F.: From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. J. of Web Semantics **1**(1) (2003) 7–26
18. S. Owre and J. M. Rushby and and N. Shankar: PVS: A Prototype Verification System. In Kapur, D., ed.: 11th International Conference on Automated Deduction (CADE). Volume 607 of Lecture Notes in Artificial Intelligence., Saratoga, NY, Springer (1992) 748–752
19. Lawford, M., Wu, H.: Verification of real-time control software using PVS. In P. Ramadge and S. Verdu, ed.: Proceedings of the 2000 Conference on Information Sciences and Systems. Volume 2., Princeton, NJ, Dept. of Electrical Engineering, Princeton University (2000) TP1–13–TP1–17
20. Srivas, M., Rueß, H., Cyrluk, D.: Hardware Verification Using PVS. In: Formal Hardware Verification: Methods and Systems in Comparison. Volume 1287 of LNCS. Springer-Verlag (1997) 156–205
21. Vitt, J., Hooman, J.: Assertional Specification and Verification Using PVS of the Steam Boiler Control System. In: Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. Volume 1165., Springer-Verlag (1996) 453–472