Formal Verification of UML Diagrams: A First Step Towards Code Generation

Jeffrey Smith[1], Mieczyslaw Kokar[2] and Kenneth Baclawski[2]

[1] Mercury Computer Systems, Inc.
[2] Northeastern University

Abstract. UML diagrams can be used for code generation. Such code should carry the meaning embedded in a diagram. The goal of this paper is to show a process in which such translation can be formally verified. To achieve this goal, the whole checking process has to be formalized. In this paper we show such a verification process and example.

Keywords - UML specification, formalization and translation, formal methods.

1 Introduction

UML diagrams are translated into code by various CASE tools. However, (1) the verification of the translation correctness is left to either the tool developers or the programmers, (2) CASE tools don't enforce a complete set of UML syntax, let alone semantics and (3) CASE tools are only capable of translating header files and constructors/destructors to a programming language. We are interested in translators that are provably correct with respect to the intended meaning of the UML language, i.e., such that preserve the intended meaning embedded in UML diagrams representing various program specifications.

Translation of UML diagrams into code may be a multi-step process [1, 2]. In order to make sure that the whole translation preserves the meaning, each of the single steps must obey such a constraint. If a specification is expressed in a formal specification language, formal methods can be used to check the correctness of such translation. However, in spite of great strides that the UML specification contributors have made in defining

semi-formal semantics, with a combination of meta-language, constraint specification and text, in the UML Semantics Guide, the UML is still not a formal language. Improvements in these semi-formal UML descriptions are needed to convey a rigorous semantics and to provide tool support to verify UML diagrams against an unambiguous specification of UML semantics. Moreover, since the UML Semantics Guide uses UML to define a *meta-model* of the UML, a formal verification process needs to be established.

In Section 2, we outline a process for formally checking the correctness of a UML diagram translator. We then give a specific example for each step of the verification process. We focus on the formalization of the UML's association and aggregation. In Section 3, we show partial formalization of these concepts in the Slang formal method language. Then, in Section 4, we show an example of a UML diagram that uses associations and aggregations as well as the result of a translation of the diagram into Slang. Finally, in Section 5, we show how to verify that an automatically generated Slang form of this UML diagram is consistent with our formalization of UML Semantics. Related UML formalization research is referred to in Section 6. In Section 7, we summarize our conclusions and point to directions for future investigations of the UML translation verification problem.

2 UML Translation Verification Process

Our UML translation verification process is shown in Figure 1. Although the Slang formal methods language [14] is used to give formal specification language examples and references, the concepts described in this paper can be attributed to any algebraic/category theory based formal language. Since UML is described in UML, Transition 1, $GT(MME)$, describes the formalization of the UML meta-model in Slang. It consists of two parts. First, we show some of the rules that we used in the formalization and then a partial formalization of the meta-model is presented. These rules significantly depend on the structure of the selected specification language (Slang). Transition 2, $TR(ME)$, shows the tool support needed to automatically translate UML applications (described as the UML Graphical

Domain) to Slang. Transition 3 represents the verification of the correctness of the translation. Here we check that instances of the Slang form of the UML Graphical Domain translation preserve the UML semantics captured in Transition 1. To explain this step, we first view both the specification of the UML meta-model and of any UML diagram as a presentation of a theory (in Slang). Our goal is to ensure that the class of models of the theory, obtained as a translation of any UML diagram, is a subclass of models of the theory of the UML meta-model (cf. [9]). In order to show this, we need to show that for each such translation there exists a morphism from the UML meta-model theory to the theory representing a given UML diagram. Transition 4, marked in Figure 1 as $Gen(ME)$, is a mapping of all possible UML diagrams that can be produced within a UML CASE tool into the UML meta-model. *Gen* maps each UML model element that appears in a particular diagram to its counterpart in the UML meta-model. We do not discuss this transition in this paper.

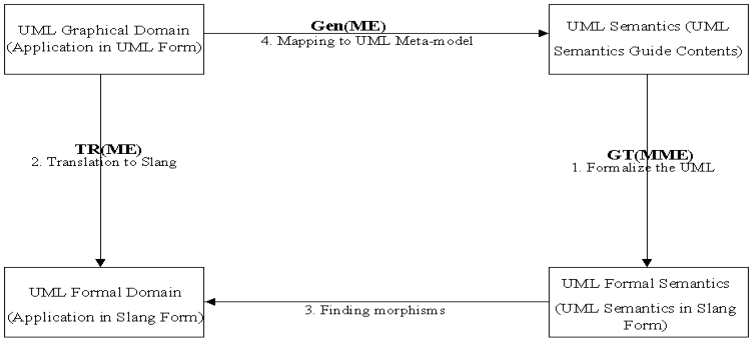

Fig. 1. Translation Verification Process

An entire UML formalization is too lengthy a topic for this paper. A complete treatment is given in [12]. We will describe a formalization of aggregation and association, not only to show an example of our process, but also to contribute a semantic distinction between these two relationships since "there is no single accepted definition of the difference between aggregation and association used by all methodologists" [8]. Figure 2 shows the subset of the UML Semantics Guide Core Package-Relationships Diagram [3], that we will address in this paper.

Our approach has been to formalize UML in category theory in a manner that closely follows the UML Semantics Guide. Alternative approaches generally formalize by mapping concepts in UML to a priori constructs such as "has-a" and "part-of" that don't necessarily follow UML very closely for two reasons: (1) UML treats aggregation as a property of one of the association ends of an association rather than as a separate modeling primitive and (2) there are several notions of UML aggregation, but just one "part-of" modeling primitive. The following sections will trace each of the steps of Figure 1 for a UML application, demonstrating how we develop a Slang form of UML semantics and verify that the translation of a UML diagram preserves the UML semantics.

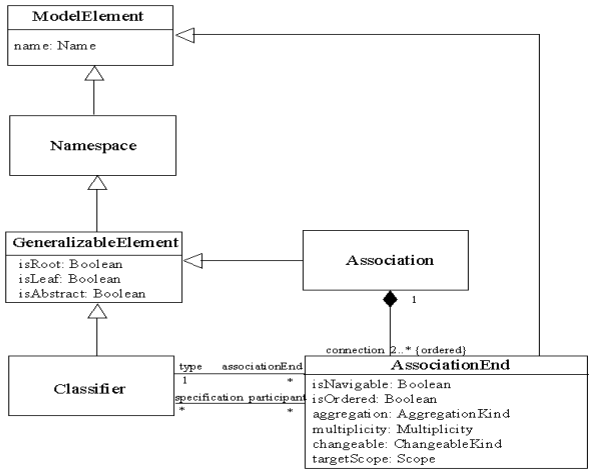

Fig. 2. Aggregation and Association Portion of UML Core Package-Relationships Diagram

3 UML/Slang Formalization

This section describes how to formalize the UML Semantics Guide in an algebraic/category theory based specification language. UML to formal specification translation is broken into a modular set of rules to break this large problem into chunks.

3.1 Specifications in Slang

The following, from [13], gives a brief overview of specifications expressed in Slang. Specifications are the fundamental objects in Slang. A specification is viewed as a presentation or description of a theory. A specification (termed in Slang as *spec*) is a finite expression that describes a potentially infinite set of strings of symbols that are within the language of a theory and a subset of this set of strings that are valid within a theory. Legal sentences are described by *signatures*. Signatures are made up of *sorts* (a declaration of the classes of objects in the specification), *ops* (ops are short for operations - a declaration of named constants that denote objects, functions and predicates of specified sorts) and *sort axioms* (an assertion of the equivalence between a primitive sort and a constructed sort). The valid sentences are specified by the *axioms* that involve both sorts and operations. Additionally, specs can include *theorems*, i.e., sentences that can be logically derived from the axioms. Specifications form a category called **Spec**. The objects in this category are specifications (specs) and arrows are such morphisms that map sorts to sorts, operations to operations and axioms to theorems.

Specifications are either given as basic specifications of these sorts, ops, axioms and theorems, or built with *translate, import* or *colimit* specification-building operations. *Translate* creates a copy of a specification, sometimes renaming some components. *Import* enriches a specification with new sorts, operations, axioms and theorems - similar to a programming language *include*. A *colimit* is used to combine specifications, taking a diagram of specifications as an input and yielding a specification that contains all the elements of the specifications in the diagram.

3.2 Background Formalization Rules

We begin with some of the background formalization rules used by the subsequent formalizations of associations and aggregations.

Object - Spec Rule. *Every Model Element in UML, specified in the UML Semantics Guide, translates to a spec containing a sort, both having the same name as the Model Element. Import a CLASS spec if the Model Element represents a class and an OBJECT spec if the Model Element represents an object. To make a lexical distinction between a spec and a sort name, the sort name begins with a capital letter followed by lower case for the remaining characters of the name, while the spec name uses all upper case letters.*

There are only two choices for an object in a specification language supporting category theory: a diagram or a spec. Of these two choices, only spec supports the *import* declarations we need to support a modular translation. The spec choice also permits more degrees of freedom than diagram because one can also associate operations (and other constructs not possible with diagrams) with a spec.

OCL Constraints to Op/Axiom Rule. *For each OCL constraint, add an associated op in the spec corresponding to the UML object that contains this OCL constraint. Specify the constraint in an axiom associated with the op.*

A constraint defines a relation. In Slang, relations are defined as ops with Boolean as their domain. For this reason, in order to specify an OCL constraint in Slang, we first have to specify a Boolean op and then specify the constraint as an axiom associated with this op.

3.3 UML Association Formalization Rule

Association - Association Instance Spec Rule. *Translate each association end of an association to a separate instance of an association end spec, filling in each of the association end constraints as ops and axioms of the association end spec. Translate each classifier of an association to*

a separate instance of an classifier spec, filling in each of the classifier constraints as ops and axioms of the classifier spec. Translate each association to a separate instance of the association spec, identifying the classifiers associated with each association end as the source and target of the association. Finally, form a colimit of these association ends, classifiers and association specs into ASSOCIATION-CLASSIFIER-COLIMIT spec, filling in the constraints, associated with association end, classifier and association relations, as ops and axioms.

The motivation to use this translation rule is to closely resemble the UML Semantics Guide, where an association is a set of tuples relating two classifiers. An association consists of at least two association ends, each of which represents a connection of an association to a classifier. This translation rule uses the UML Semantics Guide Core Meta-Model structure. As a result, associations and association ends are also translated to specs. This seems apt since we're translating objects to specs and Associations and AssociationEnds are meta-objects in Figure 2.

The formalization of the ASSOCIATIONEND, CONNECTION and ASSOCIATION specs that will be used in an example that follows, is shown in Specification 1.

Aggregation - Aggregation Instance Spec Rule. *Treat aggregation as an association, labeling the association end corresponding to the aggregate end (the side with the hollow or filled in diamond) with the type of aggregation, according to the UML Semantics Guide.*

An UML aggregation is a "part-of" relationship. If B is a part of A, then B belongs to A. This is contrasted with an association, or "is-a" relationship, where if B is associated with A, then the B-A relationship belongs to the (super) model element that includes this relationship. Nevertheless, the UML Semantics Guide does not include a meta-model element called Aggregation. Instead, it specifies aggregation as a special type of association. In UML, aggregation is an attribute of the association ends that make up an association. This attribute can take one of three possible values: *aggregate, composite* or *none*. We formalize this concept accordingly. In our Association-Association Instance translation Rule, an aggregation

is an association where one of the ends of an association connection is marked with the type of aggregation. It is marked as *aggregate* if the other end, or part, may be contained in other aggregates. It is marked as *composite* if the other end may not be part of any other composite. If it is *none*, it means the association is not an aggregation. Since our main emphasis in this paper is to show the formalization process rather than the formalization itself, not all of the details of the UML aggregation concept are included in Specification 1.

Specification 1 (Partial Formalization of the UML Meta-Model)

```
spec PAIR is
  sorts Pair, Left, Right
  op make-pair: Left, Right -> Pair
  op left:  Pair -> Left
  op right: Pair -> Right
  axiom  (equal (left (make-pair d e)) d)
  axiom  (equal (right(make-pair d e)) e)
  constructors {make-pair} construct Pair
  theorem (equal p (make-pair (left p) (right p)))
 end-spec

spec MODELELEMENT is
  sort ModelElement
  op name : ModelElement -> String
  axiom name is (fa (a: ModelElement b: ModelElement)
   (equal (name a) (name b)))
end-spec

spec NAMESPACE is
  sort Namespace
  op name : Namespace -> String
  axiom name is (fa (a: Namespace b: Namespace)
   (equal (name a) (name b)))
end-spec

spec GENERALIZABLEELEMENT is
  sort GeneralizableElement
  op name : GeneralizableElement -> String
  axiom name is (fa (a: GeneralizableElement b: GeneralizableElement)
   (equal (name a) (name b)))
  op isRoot :  GeneralizableElement -> Boolean
  axiom isRoot is (fa (a: GeneralizableElement b: GeneralizableElement)
   (equal (isRoot a) (isRoot b)))
  op isLeaf : GeneralizableElement -> Boolean
  axiom isLeaf is (fa (a: GeneralizableElement b: GeneralizableElement)
   (equal (isLeaf a) (isLeaf b)))
  op isAbstract : GeneralizableElement -> Boolean
```

```
  axiom isAbstract is (fa (a: GeneralizableElement b: GeneralizableElement)
   (equal (isAbstract a) (isAbstract b)))
end-spec

spec CLASSIFIER is
  sort Classifier
  op name : Classifier -> String
  axiom name is (fa (a: Classifier b: Classifier)
   (equal (name a) (name b)))
  op isRoot : Classifier -> Boolean
  axiom isRoot is (fa (a: Classifier b: Classifier)
   (equal (isRoot a) (isRoot b)))
  op isLeaf : Classifier -> Boolean
  axiom isLeaf is (fa (a: Classifier b: Classifier)
   (equal (isLeaf a) (isLeaf b)))
  op isAbstract : Classifier -> Boolean
  axiom isAbstract is (fa (a: Classifier b: Classifier)
   (equal (isAbstract a) (isAbstract b)))
end-spec

spec ASSOCIATION is
  import CLASSIFIER
  sort Association
  op name : Association -> String
  axiom name is (fa (a: Association b: Association)
   (equal (name a) (name b)))
  op isRoot : Association -> Boolean
  axiom isRoot is (fa (a: Association b: Association)
   (equal (isRoot a) (isRoot b)))
  op isLeaf : Association -> Boolean
  axiom isLeaf is (fa (a: Association b: Association)
   (equal (isLeaf a) (isLeaf b)))
  op isAbstract : Association -> Boolean
  axiom isAbstract is (fa (a: Association b: Association)
   (equal (isAbstract a) (isAbstract b)))
  op make-association: Classifier, Classifier -> Association
  op first:  Association -> Classifier
  op second: Association -> Classifier
  axiom  (equal (first (make-association d e)) d)
  axiom  (equal (second(make-association d e)) e)
  constructors {make-association} construct Association
  theorem (equal p (make-association (first p) (second p)))
end-spec

spec ASSOCIATIONEND is
  sorts AssociationEnd
  op isNavigable: AssociationEnd -> Boolean
  axiom isNavigable is (fa (a: AssociationEnd b: AssociationEnd)
   (equal (isNavigable a) (isNavigable b)))
  op isOrdered: AssociationEnd -> Boolean
  axiom isOrdered is (fa (a: AssociationEnd b: AssociationEnd)
   (equal (isOrdered a) (isOrdered b)))
```

```
  op name: AssociationEnd ->  String
  axiom name is (fa (a: AssociationEnd b: AssociationEnd)
   (equal (name a) (name b)))
  op aggregate: AssociationEnd -> String
  axiom aggregate is (fa (a: AssociationEnd b: AssociationEnd)
   (equal (aggregate a) (aggregate b)))
  op multiplicity: AssociationEnd -> Nat, Nat
  axiom multiplicity is (fa (a: AssociationEnd b: AssociationEnd)
   (equal (multiplicity a) (multiplicity b)))
  op changeable: AssociationEnd -> String
  axiom changeable is (fa (a: AssociationEnd b: AssociationEnd)
   (equal (changeable a) (changeable b)))
 end-spec

spec ASSOCIATION-CLASSIFIER-COLIMIT is
  import colimit of diagram
  nodes T1: TRIV, T2: TRIV, T3: TRIV, T4: TRIV,
        T5: TRIV, T6: TRIV, T7: TRIV, T8: TRIV,
        P1: PAIR, P2: PAIR, P3: PAIR, P4: PAIR,
        C1: CLASSIFIER, C2: CLASSIFIER,
        AE1: ASSOCIATIONEND, AE2: ASSOCIATIONEND, ASSOCIATION
  arcs
    T1 -> P1: {e -> Right},
    T2 -> P1: {e -> Left},
    T1 -> C1: {e -> Classifier},
    T2 -> AE1: {e -> AssociationEnd},
    T3 -> P2: {e -> Right},
    T4 -> P2: {e -> Left},
    T3 -> AE1: {e -> AssociationEnd},
    T4 -> ASSOCIATION: {e -> Association},
    T5 -> P3: {e -> Right},
    T6 -> P3: {e -> Left},
    T5 -> ASSOCIATION: {e -> Association},
    T6 -> AE2: {e -> AssociationEnd},
    T7 -> P4: {e -> Right},
    T8 -> P4: {e -> Left},
    T7 -> AE2: {e -> AssociationEnd},
    T8 -> C2: {e -> Classifier}
end-diagram
% The AssociationEnds must have a unique name within the association
  axiom OCL1 is (fa(a: AE1.AssociationEnd b: AE2.AssociationEnd)
     (implies (equal (AE1.name a) (AE2.name b)) (equal AE1.a  AE2.b)))
% At most one AssociationEnd may be an aggregate or a composite
  axiom OCL2 is (fa(a: AE1.AssociationEnd b: AE2.AssociationEnd)
   (or (implies (or (equal (AE1.aggregate AE1.a) "aggregate")
                    (equal (AE1.aggregate AE1.a) "composite"))
                    (equal (AE2.aggregate AE2.b) "none"))
       (implies (or (equal (AE2.aggregate AE2.b) "aggregate")
                    (equal (AE2.aggregate AE2.b) "composite"))
                    (equal (AE1.aggregate AE1.a) "none"))))
% The connected Classifiers of the AssociationEnds should be included
%   in the Namespace of the association
```

```
  axiom OCL3 is (fa(a: Association)
    (and (equal C1.Classifier (first a)) (equal C2.Classifier (second a))))
% No opposite  AssociationEnds may have the same name within the Classifier
  axiom OCL4 is (fa(a: C1.Classifier b: C2.Classifier)
    (implies (equal (C1.name a) (C2.name b)) (equal C1.a  C2.b)))
end-spec
```

4 UML Graphical to Formal Domain Translation

The UML Graphical to Formal Domain translation was depicted as Transition 2 in Figure 1, where we provide support to translate a UML application to a Slang spec, following the previously defined translation rules. In this section we show an example of a UML diagram (Figure 3) and parts of the spec that is the result of the translation. In the example we have a *Lecture*, which is a collection of *Student*, ordered by ID. There is a one-to-one association with a *Course*, depending on the Lecture level.

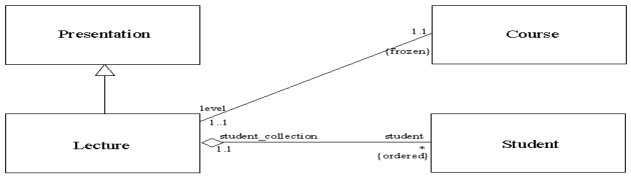

Fig. 3. UML Association and Aggregation Translation Example

The translation of this UML diagram to Slang, according to the prior semantic formalization rules, is shown below.

Specification 2 (Example: UML Association and Aggregation Translation)

```
spec LECTURE is
  sort Lecture
  op name: Lecture -> String
  axiom fa(a: Lecture) name(a) = "Lecture"
  op isLeaf: Lecture -> Boolean
  axiom fa(a: Lecture) isLeaf(a) = true
end-spec

spec STUDENT is
  sort Student
  op name: Student -> String
  axiom fa(a: Student) name(a) = "Student"
  op isLeaf: Student -> Boolean
  axiom  fa(a: Student) isLeaf(a) = true
end-spec

spec LECTURE-AE-STUDENT is
   sort Lecture-AE-Student
   op name: Lecture-AE-Student -> String
   axiom fa(a: Lecture-AE-Student) name(a) = "student_collection"
   op multiplicity: Lecture-AE-Student -> Nat, Nat
   axiom fa(a: Lecture-AE-Student) multiplicity(a) = (1,1)
   op isNavigable: Lecture-AE-Student -> Boolean
   axiom fa(a: Lecture-AE-Student) isNavigable(a) = true
   op aggregate: Lecture-AE-Student -> String
   axiom fa(a: Lecture-AE-Student) aggregate(a) = "aggregate"
   op changeable: Lecture-AE-Student -> String
   axiom fa(a: Lecture-AE-Student) changeable(a) = "none"
   op isOrdered: Lecture-AE-Student -> Boolean
   axiom fa(a: Lecture-AE-Student) isOrdered(a) = false
end-spec

spec STUDENT-AE-LECTURE is
   sort Student-AE-Lecture, n
   op name: Student-AE-Lecture -> String
   axiom fa(a: Student-AE-Lecture) name(a) = "Student"
   op multiplicity: Student-AE-Lecture -> Nat, Nat
   axiom fa(a: Student-AE-Lecture) multiplicity(a) = (1,n)
   op isOrdered: Student-AE-Lecture -> Boolean
   axiom fa(a: Student-AE-Lecture) isOrdered(a) = true
   op isNavigable: Student-AE-Lecture -> Boolean
   axiom fa(a: Student-AE-Lecture) isNavigable(a) = true
   op aggregate: Student-AE-Lecture -> String
   axiom fa(a: Student-AE-Lecture) aggregate(a) = "none"
   op changeable: Student-AE-Lecture -> String
   axiom fa(a: Student-AE-Lecture) changeable(a) = "none"
end-spec

spec LECTURE-STUDENT-AGGREGATION is
  import LECTURE, STUDENT
  sort Lecture-Student-Aggregation
```

```
  op name: Lecture-Student-Aggregation -> String
  axiom fa(a: Lecture-Student-Aggregation) name(a) =
"Lecture-Student-Aggregation"
  op isLeaf : Lecture-Student-Aggregation -> Boolean
  axiom fa(a: Lecture-Student-Aggregation) isLeaf(a) = true
  op isRoot: Lecture-Student-Aggregation -> Boolean
  axiom fa(a: Lecture-Student-Aggregation) isRoot(a) = false
  op isAbstract: Lecture-Student-Aggregation -> Boolean
  axiom fa(a: Lecture-Student-Aggregation) isAbstract(a) = false
  op make-association: Lecture, Student -> Lecture-Student-Aggregation
  op first:  Lecture-Student-Aggregation -> Lecture
  op second: Lecture-Student-Aggregation -> Student
  axiom first(make-association(d, e)) = d
  axiom second(make-association(d, e)) = e
  constructors {make-association} construct Lecture-Student-Aggregation
  theorem p = make-association(first(p), second(p))
end-spec

spec LECTURE-STUDENT-AGGREGATION-COLIMIT is
  import colimit of diagram
  nodes T1: TRIV, T2: TRIV, T3: TRIV, T4: TRIV,
        T5: TRIV, T6: TRIV, T7: TRIV, T8: TRIV,
        P1: PAIR, P2: PAIR, P3: PAIR, P4: PAIR,
        LECTURE, STUDENT, LECTURE-AE-STUDENT, STUDENT-AE-LECTURE,
        LECTURE-STUDENT-AGGREGATION
  arcs
    T1 -> P1: {e -> Right},
    T2 -> P1: {e -> Left},
    T1 -> LECTURE: {e -> Lecture},
    T2 -> LECTURE-AE-STUDENT: {e -> Lecture-AE-Student},
    T3 -> P2: {e -> Right},
    T4 -> P2: {e -> Left},
    T3 -> LECTURE-AE-STUDENT: {e -> Lecture-AE-Student},
    T4 -> LECTURE-STUDENT-AGGREGATION: {e -> Lecture-Student-Aggregation},
    T5 -> P3: {e -> Right},
    T6 -> P3: {e -> Left},
    T5 -> LECTURE-STUDENT-AGGREGATION: {e -> Lecture-Student-Aggregation},
    T6 -> STUDENT-AE-LECTURE: {e -> Student-AE-Lecture},
    T7 -> P4: {e -> Right},
    T8 -> P4: {e -> Left},
    T7 -> STUDENT-AE-LECTURE: {e -> Student-AE-Lecture},
    T8 -> STUDENT: {e -> Student}
end-diagram
  axiom OCL1 is fa(a: Lecture-AE-Student, b: Student-AE-Lecture)
     name(a) = name(b) => a = b
  axiom OCL2 is fa(a: Lecture-AE-Student, b: Student-AE-Lecture)
     ((aggregate(a) = "aggregate") or (aggregate(a) = "composite" ) =>
       (aggregate(b) = "none") or
      (aggregate(b) = "aggregate") or (aggregate(b) = "composite" ) =>
       (aggregate(a) = "none"))
  axiom OCL3 is fa(a: Lecture-Student-Aggregation)
     Lecture = first(a) & Student = second(a)
```

```
  axiom OCL4 is fa(a: Lecture, b: Student)
     name(a) = name(b) => a = b
end-spec
```

5 Verification of UML Graphical Domain Translation

As we stated in Section 2, in order to ensure the correctness of the translation we need to show that there is a morphism between the UML meta-model and the translation of a diagram. More specifically, we need to show morphisms between specs of the UML meta-model elements that are associated with any model elements of a given UML diagram through the *Gen* mapping and specs that resulted from the translation of a UML diagram.

In this section, in order to exemplify this step, we follow one thread of such reasoning. Specifically, we focus on the Lecture-Student aggregation (Figure 3). This aggregation (in UML it is an association) is mapped through *Gen* directly into the Association element of the UML meta-model and indirectly into AssociationEnd and Classifier. These meta-model elements are associated (through GT) with a number of specs in the Slang representation of the UML semantics: ASSOCIATION, ASSOCIATIONEND, CLASSIFIER, ASSOCIATION-CLASSIFIER-COLIMIT. In order to prove the correctness of a translation, we need to show morphisms from all of these specs into the specs that are the result of the translation (TR) of the Lecture-Student aggregation into its Slang representation.

To show a morphism, we need to perform two steps. First, we need to map sorts to sorts and operations to operations consistently. Then we need to map axioms of the UML meta-model specs into theorems in the translation, i.e., we need to formulate and prove appropriate theorems about the specs of the translation. The first step can be supported by the CASE tool to check the type compatibility of the mappings. For the example of the Lecture-Student aggregation, the morphisms expressed in Slang and checked by Specware are shown below.

Specification 3 (Morphisms)

```
morphism AE-LAS: ASSOCIATIONEND -> LECTURE-AE-STUDENT is
  {AssociationEnd -> Lecture-AE-Student}

morphism AE-SAL: ASSOCIATIONEND -> STUDENT-AE-LECTURE is
  {AssociationEnd -> Student-AE-Lecture}

morphism A-LSA: ASSOCIATION -> LECTURE-STUDENT-AGGREGATION is
  {Association -> Lecture-Student-Aggregation,
   Classifier -> Lecture,
   Classifier -> Student}

morphism AC-LCAS: ASSOCIATION-CLASSIFIER-COLIMIT ->
LECTURE-STUDENT-AGGREGATION-COLIMIT is
  {Classifier -> Lecture, Classifier -> Student,
   AssociationEnd -> Lecture-AE-Student, AssociationEnd ->
Student-AE-Lecture,
   Association -> Lecture-Student-Aggregation}
```

The second step, i.e., axiom mapping, is more difficult. As an example take the axiom *multiplicity* associated with AssociationEnd:

```
  op multiplicity: AssociationEnd -> Nat, Nat
  axiom multiplicity is (fa (a: AssociationEnd b: AssociationEnd)
   (equal (multiplicity a) (multiplicity b)))
```

This axiom, through morphisms $AE - LAS$ and $AE - SAL$, must map into one theorem in LECTURE-AE-STUDENT and one theorem in STUDENT-AE-LECTURE. In LECTURE-AE-STUDENT, the theorem is:

```
  theorem multiplicity-l-ae-s is (fa (a: AssociationEnd)
   (equal (multiplicity a) (1,1)))
```

The proof of this theorem is straightforward. Since the axiom states that for any a the multiplicity is $(1, 1)$, i.e., it is constant, then it is true for any substitution of variables, which includes a and b.

To complete the proof of this example, one would need to check all of the axioms of all of the associated specs. This would be a time consuming process. For this reason, we have an automatic translator TR which is verified to be provably correct [12], i.e., for any UML diagram it will produce a collection of specs for which all such morphisms as shown above exist.

6 Other Research on UML Formalization and Translation

Research on the formal semantics of the UML has been performed on many levels. Much of this research performed either UML formalization or translation (to a verifiable specification or executable language), but seldom both. For instance, in the first category, Cheng et al [4, 15, 16] have been extending their ability to construct algebraic specifications of OMT object model diagrams to UML. In Lano et al [10], an axiomatic semantic representation of UML is given in terms of theories that are used to represent classes, instances, associations and general submodels of a UML model.

The semantics of UML research was also presented in [7], where UML constraints were represented in Z. Although the Z specifications are verifiable, this research did not link directly to a CASE tool environment nor automate the specification generation. Robbins et al. [11] approach was to integrate UML with the semantics of other Architectural Design Languages (ADLs) by adding ADL extensions to UML using Stereotypes. This research generated an efficient executable language, but used a simple, partial and inconsistent view of UML meta-model, counting on other research to perform the UML formalization.

In addition to the translation from UML to Slang, Smith and DeLoach have built a translator that automatically translates from UML directly to O-Slang [6] based on an OMT theory-based object model [5] in the process of being extended to UML. O-Slang is an object-oriented form of Slang that DeLoach had built in [5]. This translation tool includes both static class and behavior (state) translation.

7 Conclusions and Future Research

In this paper, we showed a process for a formal verification of a translation of UML diagrams into a formal method language. Such a verification process is a first step of a larger task of translation of UML diagrams into provably correct code. While methods for generating provably correct

code from formal specifications are known (e.g., refinement), it is not clear how to translate UML diagrams into a formal specification language. The difficulty of this step stems from various sources: UML is not a formal specification language, UML is defined in UML, the definition provides a meta-model of UML.

Our process consists of four steps: mapping of UML diagrams into the UML meta-model, specifying the UML-metamodel in Slang, developing an automatic translator from UML diagrams to Slang, and finally proving that the translator is provably correct (i.e., for each possible translation there is a morphism from the specification of the UML meta-model to the translation of the diagram). In this paper, we outlined the process, showed partial formal specifications of the UML meta-model, showed an example of a translation of a diagram, and showed an example of a proof that would need to be carried out for each translation to verify that the translation is provably correct. We also indicated where a (formal) CASE tool can be used in the verification process.

The formalization of the UML meta-model given in this paper is a subset of a larger research effort which currently includes only part of the Foundation Package from the UML Semantics Guide. For instance, we showed how to combine specifications of meta-model elements into larger specifications (see ASSOCIATION-CLASSIFIER-COLIMIT.) The rest of the Core Package Diagram can be combined in a similar manner using the approach presented in this paper. The next step would be to include all of the UML diagrams, OCL constraints and textual descriptions of the UML semantics.

References

1. K. Baclawski, S. DeLoach, M. Kokar and J. Smith, *Object-Oriented Parsing and Transformation*, Seventh OOPSLA Workshop on Behavioral Semantics of Object Oriented Business and Systems Specifications, 1998.
2. K. Baclawski, S. DeLoach, M. Kokar and J. Smith, *Object-Oriented Transformation*, pending publication, Kluwer Publishing.
3. G. Booch, J. Rumbaugh and I. Jacobsen. *UML Semantics, Version 1.1, Rational Software Corp.*, 1997.
4. R. Bourdeau and B. Cheng. *A Formal Semantics for Object Model Diagrams*, IEEE Transactions on Software Engineering, 21(10):799-821, 1995.

5. S. A. DeLoach. *Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specifications*, PhD Thesis, Air Force Institute of Technology, June 1996, PhD Dissertation.
6. S. DeLoach, T. Hartrum and J. Smith. *A Theory-Based Representation for Object-Oriented Domain Models*, IEEE Transactions on Software Engineering, 1999. (to appear).
7. A. Evans, R. France, K. Lano and B. Rumpe *The UML as a Formal Moelling Notation*, PUML Working Group, 1998.
8. M. Fowler. *UML Distilled*, Addison Wesley, 1997.
9. J. A. Goguen and R. M. Burstall, *Some Fundamental Algebraic Tools for the Semantics of Computation, Part I: Comma Categories, Colimits, Signatures and Theories*, Theoretical Computer Science, vol. 31, 1984, pp. 175–209.
10. K. Lano and J. Bicarregui. *Formalising the UML in Structured Temporal Theories*, Seventh OOPSLA Workshop on Behavioral Semantics of Object Oriented Business and Systems Specifications, 1998.
11. J. Robbins, N. Medvidovic, D. Redmiles and D. Rosenblum. *Integrating Architecture Description Languages with a Standard Design Method*, White paper based on work sponsored by NSF grants CCR-9924846 and CCR-9701973 and DARPA, RL and USAF, University of CA, Irvine.
12. J.Smith.*UML Formalization and Transformation*, Ph.D. Thesis, Notheastern University, College of Engineering, December 1999.
13. *Specware Language Manual, Version 2.0.3*, 1998.
14. R. Waldinger et al. *Specware Language Manual: Specware 2.0.3*, 1998.
15. E. Wang, H. Richter and B. Cheng. *Formalizing and Integrating the Dynamic Model within OMT*, IEEE International Conference on Software Engineering, 1997.
16. E. Wang, B. Cheng. *Formalizing and Integrating the Functional Model into Object*