**Proceedings of the 23rd IASTED International Multi-Conference**
**SOFTWARE ENGINEERING**
**February 15-17, 2005, Innsbruck, Austria**

# USING ABSTRACT STATE MACHINES TO SUPPORT UML MODEL INSTANTIATION CHECKING

Wuwei Shen
Department of Computer Science
Western Michigan University
Kalamazoo, MI 49008, USA
email: wwshen@cs.wmich.edu

Weng Liong Low
Department of Computer Science
Western Michigan University
Kalamazoo, MI 49008, USA
email: wllow@cs.wmich.edu

**ABSTRACT**
UML is defined in terms of the four-level modeling architecture. The UML metamodel consists of informal description, class diagrams which are used to give the abstract syntax, and the well-formedness rules given in OCL. Furthermore, class diagrams have played an important role in software development because they provide basic information about a software system. In this paper, we propose a new method to support class diagrams based on Abstract State Machines. As an application of this method we will use a class diagram to represent OCL structure and therefore an OCL translator is generated. Based on the support of class diagrams and OCL, any user-defined UML model (M1-level), as an instantiation of the UML metamodel, can be compared against the UML metamodel. On the other hand, an object model (M0-level) can be checked based on the user-defined UML model (M1-level). Therefore, more errors can be reported to software developers during the early phase of software development.

**KEY WORDS**
abstract state machines, model checking

## 1 Introduction

The Unified Modeling Language has been widely accepted as a standard in software development. UML is defined in terms of the four-level modeling architecture. The UML metamodel is given by informal descriptions, class diagrams which present abstract syntax, and well-formedness rules written in the Object Constraint Language (OCL). Since the UML metamodel belongs to the M2-level, any user-defined UML model can be regarded as an instance of the UML metamodel. Comparing an M1-level UML model against the UML metamodel (M2-level) allows one to find syntactic errors in the UML model. On the other hand, a UML model is given by several diagrams. Some of them represent M1-level information while others represent M0-level information. Therefore, comparing the diagrams showing the M0-level information against the M1-level diagrams can also help software developers find errors in a software model.

Among UML diagrams, a class diagram plays an important role in software development. Although a class diagram does not reveal any dynamic aspects of a class, it provides some static information which should be valid during the whole software development. Thanks to the importance of a class diagram, the Object Management Group (OMG) uses class diagrams as a means to present the syntax in the UML metamodel. Generally speaking, a class diagram is used most when software developers design a software model.

The prime motivation of this work has been to provide a tool to find errors over populations of a variety of UML models. Supporting a class diagram in UML has become crucial in a UML tool. We support class diagrams based on Abstract State Machines. Our idea is that given any class diagram we can translate it to an ASM model, and then we can check whether an instance of this class diagram is valid or not. Particularly, due to the four-layer architecture used to define UML, we can find potential errors defined in a UML model, which can consist of several different UML diagrams other than a class diagram, after we support the UML metamodel mainly consisting of the class diagrams and well-formedness rules.

Abstract State Machines [7] was first presented by Dr. Yuri Gurevich more than ten years ago. An ASM is a state machine which computes a set of updates of the machine's variables by firing all possible updates based on the current state. The computation of a set of updates occurs at the same time and the result of computation leads to a new state to be generated.

ASMs have been applied to UML in a variety of ways. Börger et al have applied ASMs to provide semantics for UML activity diagrams and state machines ([4] and [3]). Cavarra et al have integrated UML static and dynamic views based on ASM [5]. Ober also proposed translating UML class diagrams into ASMs by defining action semantics as well as an XMI to ASM translator with manually-input OCL constraints [9]. Also, we presented a static validation method for a UML model based on XASM ([2]) [12]. Based on this method we built a tool to validate whether a UML model is statically correct [13]. From these experiences, we think ASMs can be used as a basis to support UML CASE tools.

The Object Constraint Language is used to represent some constraints in a software model, which usually cannot be represented in a UML diagram. OCL is used to give the

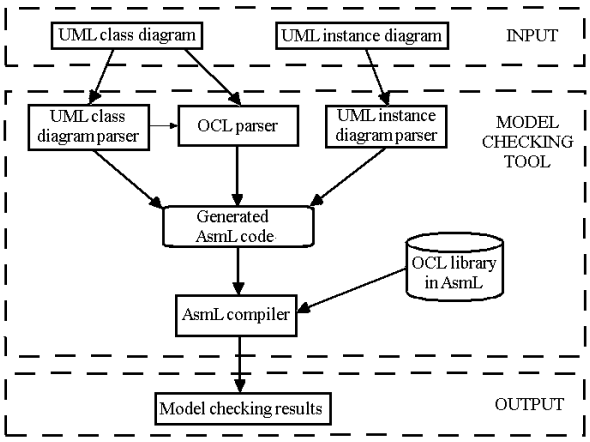


Figure 1. The model checking tool and its modules.

well-formedness rules in the UML metamodel. Since we support a class diagram based on ASMs, we use a class diagram to represent the structure of OCL types. Then the support of OCL can be partially achieved via our support of a class diagram. The implementations of operations can be added manually.

The paper is organized as follows. Section 2 introduces a class diagram and presents the implementation of a class diagram based on AsmL. Section 3 proposes a metamodel for OCL and our implementation based on AsmL. Section 4 shows an example to illustrate how to check a UML model. We draw some conclusions and give future work in section 5.

## 2 AsmL and Implementation of Class Diagrams

AsmL [8] is a high-level executable specification language developed by the Foundations of Software Engineering (FSE) group at Microsoft Research that runs on Microsoft's .NET framework. AsmL is based on the concept of abstract state machines (ASMs) and uses object-oriented programming, component-oriented programming, and functional programming. Since the UML and OCL are high-level methods of specifying a software model, they can be easily translated into a specification language like AsmL. Furthermore, the characteristics of ASMs make them useful in software verification, which is the purpose of our research in model checking of UML models.

The tool we are developing consists of four modules (Fig. 1). The first module is the UML class diagram parser - currently supporting UML models in XML Metadata Interchange (XMI) [11] format using the UML DTD - that translates the class diagram into an AsmL specification. OCL constraints are extracted from the UML model and fed an OCL parser module, which translates these constraints into an AsmL specification that depends on an OCL library module written in AsmL. The last module is an instance diagram parser that translates an instance diagram into an AsmL specification that creates and initializes instances from the specification diagram. The resulting single AsmL specification can then be compiled and executed to perform model checking on the instance diagram.

### 2.1 Diagram Translation at the Specification Level

AsmL uses an object-oriented approach while implementing ASMs. This means that classes in the UML class diagram can be mapped directly into the classes and equivalent language elements in AsmL. Of all the class diagram elements defined in the UML metamodel, he most important ones include classes, attributes, operations, associations, generalizations, and stereotypes.

Each class defined in the class diagram is translated into an AsmL class of the form

```
public class ClassName extends OclAnyImpl
```

where OclAnyImpl is a base class for user-defined models in the OCL metamodel and is defined in the next section.

Each attribute and operation in a UML class is translated into an attribute or an operation of the corresponding AsmL class respective, with the form

```
public var at_AttributeName as AttributeType
    public var op_OperationName(Param1 as
       Param1Type, ...)  as ReturnType
```

The model checking library requires access to these class members, so they are declared with public access modifiers.

Each navigable association end in a class is defined as one of the following depending on the multiplicity and the ordering of the association end:

```
public var ae_AssocEndName as AssocEndType
public var ae_AssocEndName as OclSet of
              AssocEndType
public var ae_AssocEndName as OclSequence of
              AssocEndType
```

Graphical constraints such as the multiplicity of association ends are checked by the OCL parser module before the translation of OCL constraints.

Generalizations in the class diagram are usually translated into the following:

```
public class ChildName extends ParentName
```

This is true for single-parent classes. However, since AsmL does not support multiple inheritance, only the first parent class of a child with multiple parents is directly inherited. We then copy attributes and operations of the remaining parent classes into the child class while resolving conflicts posed by multiple versions of class members. Until a future AsmL specification implements multiple inheritance, this is the workaround we are using now.

Stereotypes are used as a lightweight extension mechanism in UML to extend the metamodel. Stereotypes are translated into a child class of the base class that the stereotype extends. For example, a stereotype called *Refine_Class* that extends the meta-class *Class* in the metamodel is translated into

```
public class Refine_Class extends Class
```

because this allows the stereotype to inherit all the features and associations of the base class, similar to a generalization relationship.

## 2.2 Diagram Translation at the Instance Level

At the instance level, we assume that a corresponding class diagram, of which the current diagram is an instance, has been translated using the method described before. Usually the diagram used as input here is an object diagram at the M0 level and is an instance of the previously translated class diagram at the M1 level. However this also applies to a UML diagram (such as a class diagram, a statechart diagram, etc) at the M1 level that is an instance of a class diagram at the metamodel level (M2). For consistency we will use the term **object** to refer to the elements in the level $M_{i-1}$ diagram that are instances of the corresponding class at the $M_i$ level.

Since the given UML diagram is an instance of its corresponding class diagram, each object is translated into AsmL as an object declaration, with the type being its corresponding class at the higher level. For example, given a class *Person* in the specification level, the object *John* of type *Person* at the instance level would be translated into

```
public var John as Person = new Person ()
```

In the case of a stereotyped class at the M1-level diagram, it would be instantiated as an object of the stereotype in the metamodel. For example, a class *A* tagged with the *«Refine_Class»* stereotype would be translated into

```
public var A as Refine_Class = new
        Refine_Class()
```

Slot values and associations for the object will be assigned later in multiple steps of the ASM. An AsmL specification that contains the translated specification and instance diagrams as well as any OCL constraints can then be compiled with the OCL library and executed to verify that the instance diagram satisfies all OCL constraints given in the specification class diagram.

# 3 OCL Metamodel and Its Implementation

The OCL is used to represent constraints in the UML class diagram. Therefore, we also parse OCL constraints and translate them into AsmL in order to enforce these constraints on the instance-level diagrams described in the previous section. There has been previous work [6] to write OCL compilers to translate OCL into languages such as Java. Here we translate OCL into AsmL to integrate it with the AsmL translation of class diagrams. To do this, we use a class diagram to represent the structure of OCL types, similar to the one presented in [1]. The OCL library is generated from the OCL metamodel by applying the translation schema in the previous section, and the various method implementations are added manually based on the operation semantics.

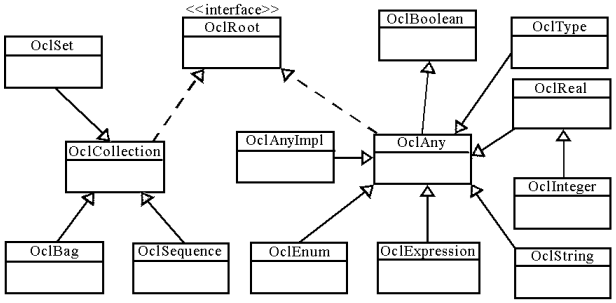


Figure 2. Class diagram for OCL type hierarchy.

## 3.1 Overview of the OCL Metamodel

Fig. 2 shows the class diagram for the OCL type hierarchy. As a general rule, all the types in OCL (from basic types to collection types to user-defined types or classes in the UML class diagram) are translated into AsmL classes. All basic data types inherit from the *OclAny* class, which is the supertype of all OCL types. The class *OclType*, which contains information about a particular type, is a subclass of *OclAny*. Collection types in OCL inherit from an *OclCollection* class. The *OclCollection* and *OclAny* classes implement the *OclRoot* interface, which contains features and additional operations common to all the OCL types. Another subclass of *OclAny*, *OclAnyImpl*, is the base class for all user-defined types specified in a UML class diagram. It contains attributes and methods to deal with user-defined types in relation to the entire OCL type hierarchy.

The AsmL classes for most of the basic OCL types and collection types encapsulate the corresponding types already available in AsmL. However, the *Real* basic type encapsulates the AsmL *Double* type while the *Bag* collection type encapsulates the AsmL *Seq* (Sequence) type. The latest version of the AsmL compiler supports a native *Bag* type, but it breaks our existing implementation and there is little documentation for it right now. The collection types are implemented as type classes in AsmL. For example, *OclCollection* is defined as

```
public class OclCollection of T implements
        OclRoot
```

```
public sealed class OclBoolean extends OclAny
   public var val as Boolean
   public boolAnd (b as OclBoolean) as OclBoolean
      return ToOclBoolean (val and b.val)
   public boolOr (b as OclBoolean) as OclBoolean
      return ToOclBoolean (val or b.val)
   public boolXor (b as OclBoolean) as OclBoolean
      return ToOclBoolean ((val or b.val) and
         not (val = b.val))
```

Figure 3. Basic type operations

This means that each collection class can only contain one type, T, of objects, although T can be the AsmL class *Object*, which is the superclass of every other AsmL type.

## 3.2 Basic Type Operations

Simple wrapper methods are used to encapsulate the primitive AsmL type operations that are defined in the OCL specification [10]. Operation names are retained except for basic arithmetic *Boolean* operations since they conflict with AsmL's keywords. An example is shown in Fig. 3.

## 3.3 Collection Type Operations

The collection types in OCL present the greatest challenge in our AsmL representation of OCL because the *OclSet*, *OclSequence*, and *OclBag* types have a common *OclCollection* base class, yet have different internal representations. Our work is simplified by the fact that the OCL *Bag* and *Sequence* types map to the same AsmL *Seq* type, with the reasons given in the previous section. Therefore, the member variable declared in *OclCollection* uses AsmL's disjunctive type, which allows a variable to have values for two or more types. Therefore, we only need to map three OCL types to two AsmL types. The following is the member declaration:

```
public var coll as Set of T or Seq of T
```

*OClCollection* operations cast the disjunctive type into the proper AsmL collection type in order to use specific *Set* or *Seq* operations (eg. Fig. 4).

```
public size() as OclInteger
  if isSet() then
    return ToOclInteger((coll as Set of T).Size)
  else
    return ToOclInteger((coll as Seq of T).Size)
```

Figure 4. OclCollection operations

Most of the collection operations are easy to implement in AsmL because AsmL already has the basic collection types. Therefore most of the methods in these collection classes are simply wrapper methods for the native collection operations. However, certain collection operations,

```
public attributes() as Set of String
   var s as Set of String = {}
   var f = objectType.GetFields (
      (System.Reflection.BindingFlags.Public +
      System.Reflection.BindingFlags.NonPublic +
      System.Reflection.BindingFlags.Instance) as
      System.Reflection.BindingFlags)
   step foreach x in (f as PrimitiveArray of
     System.Reflection.FieldInfo)
      let temp = x.Name as String
      if temp.StartsWith ("at_") then
         add (temp.Substring (3) as String) to s
   step
      return s
```

Figure 5. OclType attributes() operation

such as `forAll`, `exists`, `select`, `reject`, `collect` and so on, take OCL expressions as arguments. We describe how this issue is dealt with below.

## 3.4 Iterate-based Collection Operations

A number of collection operations in OCL have OCL expressions as parameters. Some examples include *reject*, *select*, *forAll*, *exists*, and *collect*. In this paper we call them iterate-based collection operations because they can be described in terms of a generic *iterate* operation that evaluates a given OCL expression for each of the elements in a collection. It is defined as

```
collection→iterate( elem :  Type; acc :  RetType
   = InitValue | expression-with-elem-and-acc )
```

This operation is powerful because we can reduce most of the iterate-based operations into the *iterate* form and then apply the same translation schema into AsmL. The pseudocode for the general form of the *iterate* operation is:

```
step
   initialize accumulator
step foreach elem in collection
   evaluate OCL expression on elem
   update accumulator with result of expression
```

Each iterate-based expression has one accumulator variable that is visible to all statements in the translated code block. Indentation is AsmL is important, so care as to be taken so that indentation in the translated AsmL specification is correct. In the case of nested collection operations, the resulting AsmL statements is nested in the proper order so that the identifiers are in the correct scope. With this schema, the OCL collection operations, one of the major features of OCL, can be translated into AsmL for execution in model checking.

## 3.5 Type Information

Type information in AsmL is obtained by using the reflection capabilities of the .NET framework defined in the System.Reflection namespace. They are used in the various

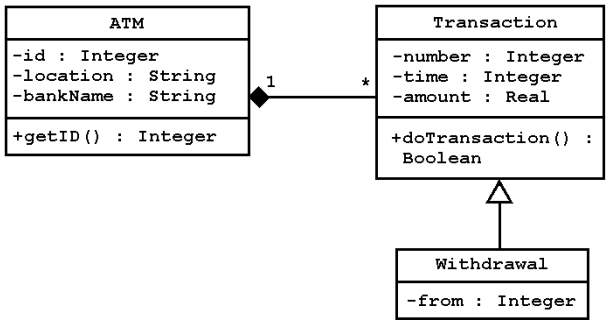


Figure 6. Simple class diagram for an ATM model.

*OclType* operations. However, due to the limitations of the current reflection classes, type information for elements of a collection type is lost. Fig. 5 shows the AsmL code for the *OclType* operation `attributes()`.

### 3.6 Model Checking Code

We need a way to execute the translated OCL constraints. To do this, the translated OCL constraints are placed in a `verify()` method that is a member of each translated AsmL class corresponding to the classes in the UML class diagram. Each constraint is executed in separate steps of the ASM that comprises the `verify()` method. This method returns a true value if and only if all OCL invariants for a particular object of a class is satisfied. When the object diagram is translated, the tool writes a `Main` function that first instantiates each object in the diagram with its initial values and links. In the next ASM step, the `Main` function calls the `verify()` method for each object to perform the checking.

## 4 Case Study: ATM example

In this section we present a simple model of an ATM and show its translation into AsmL (Fig. 6). We also show how the resulting AsmL specification can be used in conjunction with the OCL library to verify that OCL constraints are satisfied when an object diagram is given. Fig. 7 is the translated AsmL specification representing the three classes.

Let us also assume that the OCL constraints in Fig. 8 are given in the model. The constraints for the *ATM* class limits the ID range and ensures that transaction numbers increase with time. Each *Transaction* instance must also have a positive transaction number.

Up until the translation of OCL constraints, the generated AsmL specification contains only declarations and definitions of class and methods, which can be compiled but not executed. If an object diagram is given as input to the tool, a `Main` function is generated and appended to the end of the AsmL specification.

As an example, assume that the object diagram in

```
public class ATM extends libOcl.OclAnyImpl
   public var at_id as libOcl.OclInteger
   public var at_location as libOcl.OclString
   public var at_bankName as libOcl.OclString
   public var ae_transaction as libOcl.OclSet
      of Transaction
   public op_getId() as libOcl.OclInteger

public class Transaction extends
  libOcl.OclAnyImpl
   public var at_number as libOcl.OclInteger
   public var at_time as libOcl.OclInteger
   public var at_amount as libOcl.OclReal
   public var ae_atm as ATM
   public op_doTransaction() as libOcl.OclBoolean

public class Withdrawal extends Transaction
   public var at_from as libOcl.OclInteger
```

Figure 7. Translation of ATM model.

```
context ATM
  inv TransIncrWithTime: self.transaction->
    forAll (x,y | x.number < y.number implies
    x.time < y.time)
  inv ValidIDRange: self.id > 0 and self.id < 100

context Transaction
  inv PositiveTransNum: self.number > 0
```

Figure 8. OCL constraints for the ATM model.

Fig. 9 is input into the tool. We can see that the *trans1* object has a value of -30 for *number*, which violates the `PositiveTransNum` constraint. The *atm1* object also violates the `TransIncrWithTime` constraint since the transaction numbers of the *Transaction* objects do not increase with time. When the generated AsmL specification is compiled with the OCL library and run, it produces the error message shown in Fig. 10, showing the OCL constraints that are violated.

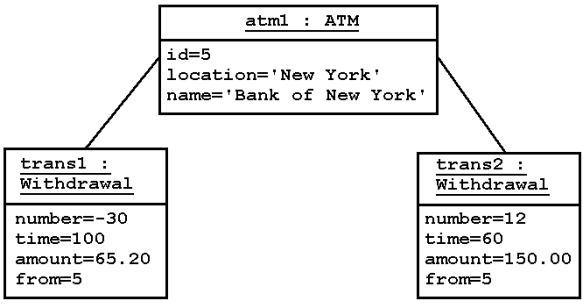


Figure 9. Object diagram that is not a valid instance of the corresponding class diagram.

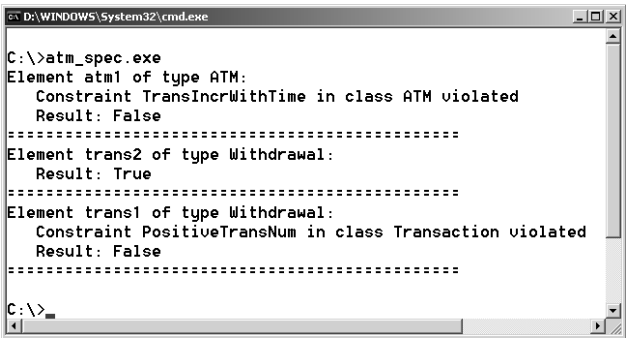


Figure 10. Error message returned while verifying object diagram.

## 5 Conclusion and Future Work

In this research we propose to use AsmL to implement UML class diagrams. As an example, we use a class diagram to represent the OCL structure, i.e. the OCL metamodel and then an AsmL specification for OCL can be generated. After adding the AsmL specification for operations defined OCL, a complete OCL parser based on AsmL is implemented.

Since we can support class diagrams and OCL, any user-defined model (M1-level) can be compared against the UML metamodel and some syntactic errors can be found. In addition, within a user-defined model, the diagrams showing M0-level information can be compared against the diagrams showing M1-level information to find some errors in the model.

As an application of this research, we are working on expanding the UML metamodel to support software refinement. After introducing new stereotypes in class diagrams and well-formedness rules based on software refinement, we can help software developers to check whether two consecutive software models during software refinement are semantically equivalent. Our motivation for this research is to build a UML CASE tool that can help software developers find not only errors in one level of software model but also errors occurring at different levels during software refinement.

**Acknowledgement.** The authors would like to thank James Huggins, Kevin Compton, and the FSE group at Microsoft for their valuable support and help during the writing of this paper.

## References

[1] Dresden OCL toolkit. http://dresden-ocl.sourceforge.net/.

[2] Matthias Anlauff. XASM- an extensible, component-based abstract state machines langu age. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer.

[3] Egon Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000*, volume 1816 of *LNCS*. Springer.

[4] Egon Börger, A. Cavarra, and E. Riccobene. Modeling the dynamic of UML state machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of *LNCS*, pages 223–241. Springer.

[5] A. Cavarra, E. Riccobene, and P. Scandurra. integrating uml static and dynamic views and formalizing the interaction mechanism of uml state machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *ASM 2003*, volume 2589 of *LNCS*, pages 229–243. Springer.

[6] Frank Finger. *Design and Implementation of a Modular OCL Compiler*. PhD thesis, Dresden University of Technology, 2000.

[7] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[8] Microsoft FSE Group. Introducing AsmL: A Tutorial for the Abstract State Machine Language. Technical report, Microsoft FSE Group, Dec, 2001.

[9] Ileana Ober. An ASM Semantics of UML Derived from the Meta-model and Incorporating Actions. In *Proceedings of Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM 2003*, volume 2589 of *LNCS*, pages 356–371. Springer, 2003.

[10] OMG Unified Modeling Language Specification, version 1.5, March 2003.

[11] OMG XML Metadata Interchange (XMI) Specification. OMG. 2000.

[12] Wuwei Shen. *The Application of Abstract State Machines in Software Engineering*. PhD thesis, Dept of EECS, the University of Michigan, September, 2001.

[13] Wuwei Shen, Kevin Compton, and James Huggins. Formalizing UML state machines for model checking. In *Proceeding of 26th annual international computer software and applications conference, Oxford University*, pages 147–152, IEEE Computer Society, August 2002.