A Formal Analysis Framework of Object-Oriented Designs

Boumediene Belkhouche[*] and Sonal Dekhane[**]

[*]Faculty of Computer Science, Tulane University, New Orleans, LA, USA
bb@eecs.tulane.edu

[**]Doctoral Candidate, Tulane University, New Orleans, LA, USA
dekhane@eecs.tulane.edu

ABSTRACT

This paper describes a formal framework for specification and analysis of object-oriented designs. The formal design notation models both the structural and behavioral views of the design. The analysis framework supports a Goal Expression Language (GEL) that allows the user to express his/her analysis goals for the specific design under consideration and the processor then analyzes the design according to these goals. Code generation is supported following successful analysis.

Keywords: *object-oriented design, design specification, formal analysis, goal expression language*

1. INTRODUCTION

Detecting errors at earlier stages of software development is extremely crucial to avoid time and budget related problems. Also since the most constant quality of software is change, it is important that the design follows all the OOD principles. Before the design goes into the implementation stage, it has to be analyzed, so that errors, if any can be detected. To perform rigorous analysis of a design, the analysis tools require that the design notation used for the specification of the design be precise and formal. Research has been done in this area, but there is a lack of a framework that provides a design notation and an analysis framework that supports analysis based on user defined constraints. The popular design notations present today do not provide sufficient formality. Existing work centers around analysis tools that detect inconsistencies among different designs or perform analysis based on predefined constraints. Our research addresses these issues. GEL provides flexibility to the user to specify analysis constraints. This makes the analysis more specific to that particular design. Instead of hard coding the analysis constraints into the analysis tool, our processor supports user-defined analysis goal execution. This makes the analysis framework more stable against evolving design notations. Finally, after successful analysis, C++ code is generated.

The paper is organized as follows. Section 2 discusses some of the issues associated with object-oriented analysis. In section 3 a formal design notation is presented. In section 4 a brief overview of our formal analysis framework and GEL is presented. Section 5 describes the transformation from GEL to intermediate level of the language processor that performs rigorous analysis of the design. In section 6 the code generation process is described. Section 7 summarizes this work and suggests directions for future research.

2. ISSUES AND OBJECTIVES

UML is an excellent visualization tool that provides different views of the same system thereby providing more information about the system [3]. It is popular today because it is extremely intuitive. The problem though is that it is not completely formalized [7]. The semantics of UML is expressed in natural language. The diagrams and the informal semantics are subject to an individual's interpretation and can lead to disagreement between individuals. The constraints are expressed using Object Constraint Language (OCL). The semantics of this language are yet to be completely precisely defined [12]. Due to the lack of formality and precision in UML, analysis tools based on UML cannot support rigorous analysis of a design.

Other formal notations like Syntropy, LOTOS and Object Z were developed to formalize UML. A large semantic gap existed between the formal notations and the graphical ones that they represented. In reference [4] a formal framework for UML state diagrams using ASMs is proposed. In [9] the UML model and OCL constraints are translated to the language of theorem prover PVS. Thus along with defining formal syntax and semantics for both UML and OCL, it also supports formal verification of systems. This work deals with class diagrams and state diagrams. As noted in [9] the coarse level specification of OCL constraints is not sufficient to automate the verification process. A number of analysis techniques that were developed, detected inconsistencies among different diagrams ([8], [5], [6]). These techniques are good at detecting inconsistencies, but can let some design flaws go unnoticed, unless the designer notices them. In [7] the authors discuss a framework that performs analysis of the design by proposing a conjecture and proving it true. Since, the basis of this is still a conjecture; it can hide some design flaws. Other testing tools were developed to test the behavior of the system ([10], [11]). These are rather involved techniques and we believe that more comprehensive results can be obtained by

using simpler techniques. Other analysis tools analyzed the designs based on predefined constraints [2]. This raises a couple of issues. First, this kind of analysis is more generalized and the constraints apply to a broad range of designs. Finer details of the design cannot be analyzed by such hard-coded constraints. These finer constraints are specific to each design and can vary from design to design. Also, as the design notation evolves, the predefined constraints may no longer be of any use and a new tool might have to be created.

This paper addresses these issues and provides a formal framework for the specification and analysis of the design. The main objectives of this research are as follows:

- To develop a formal notation for design specification using simple set theory.
- To design a simple language for the user to express analysis goals.
- To develop a formal analysis framework to perform rigorous analysis of the design according to the specified constraints.
- To support code generation after successful analysis.

3. FORMAL NOTATION

Our goal is to define a simple and intuitive design notation that can express all elements of an OOD. Our notation is a set based notation and uses previously defined concepts in Object Oriented Design Language (OODL) and Communicating Sequential Processes (CSP). OODL is an expressive language that captures the notations of Booch's and Coad's approaches. This language was formally defined in [1]. The behavior modeling is based on the concept of Object Life History (OLH) and CSP. CSP models both the OLH and the object interactions. The structural model consists of a design and the behavior is a part of this design. The behavior model consists of objects. Objects interact with each other using communication events that are either outgoing or incoming and interact with the environment using general events. The OLH describes the different sequences of these events that can occur, thus capturing the dynamic behavior of the system. The design is defined as a set of elements, wherein each element in turn is a set. A high-level description of the formal notation is discussed below.

- The root component of our formal notation is a global system, which is defined as a set of packages.
- Each package in turn has a package name, a set of dependencies in which this package participates, a set of classes and a set of relationships between these classes.
- Each class now has a class name, a set of attributes, a set of methods and a behavior.
- Each dependency in the dependency set has two elements which are the names of the two packages participating in the dependency.
- Each relationship in a relationship set has a name, the two participating classes, role names and their respective cardinalities.
- Each attribute has a name and a type along with its access modifier, which can be public, private or protected.
- Each method has a name, an access modifier, a return type and a set of parameters, where each parameter in turn has a name and a type.
- The behavior set has a behavior name, a set of two events that can participate in that class and a set of sequences of events that can occur.
- Each event has a name and a type which can be general, incoming or outgoing.
- Each sequence is a set of events that occur in a particular order represented by edges.

4. FORMAL ANALYSIS

Once the design elements have been defined formally, rigorous analysis can be done on the design to detect errors. To analyze the design according to the user's analysis goals a simple Goal Expression Language (GEL) is developed. The goal is to keep this language as simple and as close to natural language as possible, while making it powerful enough to express various types of constraints. The processor includes support for this language to perform semantic analysis of the design. The basic element of this language is an analysis goal for which a design needs to be analyzed. This goal can be expressed using any of the following elements of the language.

- Set membership element: The analysis goal is applied to some of the design elements like class, method etc. Members of these elements or sets need to be identified first.
- Conditional element: Filtering conditions can be used for selecting specific members of the set.
- Iterator element: Iterations can be specified on set members by combining the universal quantifier and/or the conditional element.
- Constraint element: Once the members are identified, the constraints need to be specified. These constraints can be specified using:
    - Relational operators
    - Set inclusion/exclusion operators
    - Existential quantifier
    - Universal quantifier
    - If-then element
    - Logical operators to specify more than one constraint
    - Predicates can be applied to set members like cycles, superclass, subclass etc.

Some commonly used properties of OODs are used as examples to demonstrate the use of the language.

4.1. UNIQUE IDENTITY CONSTRAINTS

Some of the elements are required to be unique in a design. Class names in a package cannot be repeated. The same can be applied to attributes in a class and parameters in a method, but the same does not apply to methods. The identity of a method is not only its name but also its parameters. So method signatures on the

whole have to be unique. Similar identity constraints can be applied to relationships.

1. *Class names in a package should be unique*
   each class c1, c2 in each package p
   name(c1) != name(c2)
2. *Method signatures in a class should be unique*
   each method m1, m2 in each class c
   m1 != m2

Discussion: The design elements on which the constraints should be applied have to be specified along with an instance of that element, like *class c1, c2*. The constraints can be applied either directly to these instances as a whole or on each field like *name* of class, *type* of attribute etc. The keyword each specifies that all the instances in each element have to satisfy these constraints.

4.2. CONSTRAINTS ON ATTRIBUTES

1. *All attributes in a class should be private*
   each attribute a in each class c
   visbility(a) == private
2. *Attribute type should be built-in or user defined*
   each attribute a in each class c
   type(a) ∈ primitive ∨
   type(a) ∈ userdefined

Discussion: primitive, userdefined and private (or any other access modifier) are all defined as keywords. primitive includes int, float, char, bool and string since a is an attribute. userdefined specifies the namespace for attributes of each class. Similar constraints could be applied to parameters, except that they cannot have an access modifier.

4.3. CLASS ISOLATION CONSTRAINT

This constraint specifies that no class in a package can be isolated. So, it has to participate in at least one relationship in that package.

1. *All classes in a package should have some relationship with at least one of the other classes in the package*
   each class c in each package p
   ∃ relationship r in same package p such that
   superclass(r) == name(c) ∨
   subclass(r) == name(c)

Discussion: The aggregation, composition and generalization relationships identify participating classes as superclass and subclass, which are used as predicates in our language. The keyword *same* specifies that the same set as used earlier is considered.

4.4. RELATIONSHIP CONSTRAINTS

Relationships between classes can be identified using their name, the two classes that participate in that relationship and their cardinality. The same pair of classes cannot appear more than once in relationships. Also, both the classes participating in a relationship have to be different. Moreover in a composition relationship, the part class or the subclass cannot be a "part" in another relationship.

1. *Relationship pair(c1, c2) can appear at most once in a package*
   each relationship r1, r2 in each package p
   superclass(r1) != superclass(r2) ∧
   subclass(r1) != subclass(r2)
2. *The two classes participating in a relationship have to be different*
   each relationship r in each package p
   superclass(r) != subclass(r)
3. *In composition the "part" class cannot be "part" of any other relationship*
   each relationship r1, r2 in each package p such that name(r1) == composition AND name(r2) == composition
   subclass(r1) != subclass(r2)
4. *Circular inheritance cannot exist in a package*
   all relationship r in each package p such that name(r) == generalization
   !cycles(r)
5. *Cardinality should be non-negative integers*
   each relationship r in each package p
   cardinality1(r) >= 0 ∧
   cardinality2(r) >= 0 ∧
   cardinality1(r) ∈ integers ∧
   cardinality2(r) ∈ integers

Discussion: Conditions can be specified for selecting instances using keywords *such that*. *cycles* is a predicate that can be used to specify cycles constraint on relationships, packages or events.

4.5. EVENT CONSTRAINTS

These constraints specify that the incoming events in one class should have corresponding outgoing events in the same class from which this event was received. Similarly an outgoing event in a class should have a corresponding incoming event in another class that is being referenced as receiving this event.

1. *Every incoming event has a corresponding outgoing event*
   each event e in each behavior b, class c
   if type(e) == incoming then
   ∃ event e1 in behavior b1, class c1
   such that name(e) == name(e1) ∧
   type(e1) == outgoing
2. *Every outgoing event has a corresponding incoming event*
   each event e in each behavior b, class c
   if type(e) == outgoing then
   ∃ event e1 in behavior b1, class c1
   such that name(e) == name(e1) ∧
   type(e1) == incoming
3. *Every event has a corresponding method of the same name in its structural diagram*
   each event e in each behavior b, class c
   ∃ method m in same class c such that
   name(e) == name (m)
4. *The behavior name is the same as the class name in the structural diagram*
   each behavior b in each class c
   name(b) == name(c)
5. *All events in each sequence must be declared*
   each event e in each sequence s, behavior b
   ∃ event e1 in same behavior b such that

e1 == e

6. *All events declared must be used in some sequence*
   each event e in each behavior b
   ∃ sequence s in same behavior b
   such that some(event(s)) == e
7. *No sequence can start with an outgoing event*
   each sequence s in each behavior b
   type(first(event(s))) != outgoing

Discussion: *if* statements can also be used as usual. Logical operators can be used to combine conditions as well as *then* statements.

5. TRANSLATION

This section discusses the mapping from the analysis expression language to the intermediate language of the processor.

- The "each" statements map to iterations that are performed on each of the objects of the specified design elements. The included operations are performed on each of the objects.
  E.g. each class c in each package p
  Stmt1
  MAPS TO: In each package p, consider each class c and execute Stmt1 on c.
- While the keyword "each" means perform the following operation on each object, the keyword "all" means, consider a set of all the objects and perform the operation on this set.
  E.g. all relationship r in each package p such that name(r) == generalization
  Stmt1
  MAPS TO: Consider a set of all relationships with name "generalization" and execute Stmt1 on this set.
- If the operations are not to be performed on all the objects, filtering conditions can be specified using "such that" constraint. Hence, "such that" statements map to filtering conditions.
  E.g. each relationship r in each package p such that name(r) == generalization
  Stmt1
  MAPS TO: In each package p, consider each relationship r with name generalization and execute Stmt1 on r.
- Relational operators can be specified for comparing predicates on objects like name, type etc. They can also be used to compare objects themselves, in which case each field of one object is compared to corresponding field of the other object.
  E.g. name(c1) != name(c2)
  MAPS TO: c1.Name != c2.Name
  E.g. m1 != m2; If m1 and m2 are methods
  MAPS TO: m1.Type != m2.Type OR
  m1.Name != m2.Name OR
  m1.Parameter != m2.Parameter
- Relational operators can also involve keywords. The keywords map to function calls. These functions are predefined.
  E.g. a IN primitive
  MAPS TO: a belongs to the predefined set
  a ∈ {int, float, char, bool, string}; if a is an attribute or a parameter
  a ∈ {int, float, char, bool, string, void}; if a is a method
- "exists" statements map to search functions meaning search for at least one element that satisfies the condition.
  E.g. ∃ event e in each class c such that
  Stmt1
  MAPS TO: search in class c at least one event e such that it satisfies condition specified in Stmt1. Multiple conditions can be specified using logical operators ∧ and ∨.

6. CODE GENERATION

The design can be analyzed for different constraints and the model can be refined in case of any errors. After successful analysis of the design, C++ code is generated. This code is the skeleton based on the design, implementing both the structural and behavioral models of the design. The structural code is the class definition, while the behavioral code is the trace of events specified in the behavior model. A simplified model of a university system is shown in the Appendix.

7. CONCLUSION

A formal framework for the specification and analysis of OODs was developed. This framework provides a formal, set-based design notation, a goal expression language, an analysis processor and a code generator. The design notation models both structural and behavioral views of the design and is semantically similar to the graphical notations it models. GEL and the analysis processor at this time analyze static behavior of the system. More work can be done to include support for dynamic behavior analysis of the system. The language is powerful enough to express the dynamic behavior goals. The code generator needs to handle the dynamic behavior goals. The formal semantics of GEL will be explained in detail elsewhere. Work is currently being done on it.

REFERENCES

[1] Belkhouche B., and Chavarro M., "Analysis of Object-Oriented Designs," *Journal of Object-Oriented Programming*, pp. 30-42, Feb 1995.

[2] Belkhouche B., and Nix A., "Formal Analysis of UML-based Designs," *in Proceedings of the International Conference on Software Engineering Research and Practice, SERP '04*, USA, pp. 220-226, Jun 2004.

[3] Booch G., Rumbaugh J., and Jacobson I., *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[4] Borger E., Cavarra A., and Riccobene E., "On Formalizing UML State Machines using ASMs," *Information and Software Technology*, vol. 46, no. 5 pp. 287-292, 2004.

[5] Egyed A., *Automatically Validating Model Consistency During Refinement*, Technical Report, University of Southern California, 2000.

[6] Engels G., Groenewegen Luuk., Heckel R., and Kuster J., "A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models," *in Proceedings of 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 186-195, 2001.

[7] Evans A. S., France R. B., Lano K.C., and Rumpe B., "The UML As A Formal Modeling Notation," *UML '98-Beyond the Notation*, Mulhouse, France, pp. 75-81, 1998.

[8] Fradet P., Metayer D. L., and Perin M., "Consistency Checking for Multiple View Software Architectures," *Lecture Notes in Computer Science*, pp. 410-428, 1999.

[9] Kyas M., Fecher H., deBoer F., Jacob J., Hooman J., Zwaag M., Arons T., and Kugler H., "Formalizing UML Models and OCL Constraints in PVS," *Electronic Notes in Theoretical Computer Science*, pp. 39-47, 2005.

[10] Moreira A., and Clark R., "Combining Object-Oriented Modeling and Formal Description Techniques," *in Proceedings of 8th European Conference on Object-Oriented Programming (ECOOP'94)*, pp. 344-364, 1994.

[11] Pilskalns O., Andrews A., Ghosh S., and France R., "Rigorous Testing by Merging Structural and Behavioral UML Representations," *in Proceedings of* the 6th International Conference on the Unified Modeling Language, pp. 234-248, 2003.

[12] Sendall S., and Strohmeier A., "Using OCL and UML to Specify System Behavior," *Object Modeling with the OCL, The Rationale Behind the Object Constraint Language*, pp. 250-279, 2002.

APPENDIX

University System Design

```
//University System Model
begin design UniversitySystem
begin package UniversitySystem
class Department
 attributes:
   private deptId: int,
   private deptName: string
 relations:
   association Student enrolls [1..*],
   association Course offers [1..*]
 operations:
   public AddDept(int: DId, string: DName): bool,
   public DelDept(int: DId): bool
end Department

class Student
 attributes:
   private studentId: int,
   private FName: string,
   private LName: string,
   private DId: int
 relations:
   association Department majorsIn[*..1],
   association Registration requests[1..*]
 operations:
   publicAddStudent(int: studentId, string: FName,
   string: LName, int: DId): bool,
   public DelStudent(int: studentId): bool
end Student

class Course
 attributes:
   private int: courseId,
   private string: courseName,
   private int deptId,
   private int creditHours
 relations:
   association Department belongsTo[*..1]
   association Registration requires[1..*]
 operations:
   public AddCourse(int: CId, string: CName, int: DId,
   int: CHours): bool,
   public DelCourse(int CId, int DId): bool
end Course

class Registration
 attributes:
   private int: SId,
   private int: CId
 relations:
   association Student registers[*..1],
   association Course refersTo[*..1]
 operations:
   public AddRegistration(int: SId, int: CId): bool,
   public DelRegistration(int: SId, int: CId): bool
end Registration

begin behavior UniversitySystem
OLH Department
alphabet Department = {AddDept(did, dname),
DelDept(did), Course ! AddCourse(cid, cname, did,
chrs), Course ! DelCourse(cid)}

Department = (AddDept(did, dname) -> Course !
AddCourse(cid, cname, did, chrs) | Course !
DelCourse(cid, did) -> DelDept(did))

end behavior UniversitySystem
end package UniveristySystem
end design UniversitySystem

//Generated Code Structural
class Department
{
 private:
   int deptId;
   string deptName;
 public:
   Department();
   ~Department();
   bool AddDept(int Did, string DName);
   bool DelDept(int DId);
};
```

```
class Course
{
 private:
   int courseId;
   int courseName;
   int deptId;
   int creditHours;
 public:
   Course();
   ~Course();
   bool AddCourse(int CId, string CName, int Did, int
   CHours);
   bool DelCourse(int Cid, int DId);
};

//Generated Code Behavioral
behavior Department
{
   AddDept(did, dname);
   AddCourse(cid, cname, did, chrs);

   OR

   DelCourse(cid, did);
   DelDept(did);
}
```