Analysis Goal Expression Language

Boumediene Belkhouche
Tulane University, Department of Electrical Engineering and Computer Science
New Orleans
Louisiana, USA
bb@eecs.tulane.edu

Sonal Dekhane
Tulane University, Department of Electrical Engineering and Computer Science
New Orleans
Louisiana, USA
dekhane@eecs.tulane.edu

ABSTRACT

This paper presents the abstract representation of Goal Expression Language (GEL). GEL is a very simple language used to specify analysis goals for an object-oriented design. This language is close to natural language and hence the learning factor is very low. GEL is part of a formal framework designed for the specification and analysis of object-oriented designs (FSAF). GEL can be used to express analysis goals for the static and dynamic behavior analysis of the design. This paper describes the basic components of GEL, how and where they can be used and also provides examples for demonstration. This abstract representation is the first step towards the formal semantics of GEL.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Object-Oriented Technology

General Terms
Analysis of OODs

Keywords
object-oriented analysis, goal expression language, GEL informal semantics, GEL abstract syntax

1. INTRODUCTION

In software development analyzing the software design and detecting errors early on can prove extremely useful in saving trouble at later stages of development. Rigorous analysis demands formal specification of the design. The analysis techniques should be independent of the underlying specification technique used. A good analysis technique should also allow the designer to specify his/her analysis goals for the design, so as to be able to identify the most subtle design flaw. Today’s widely used design tool UML lacks formality that is required for rigorous analysis. Our previous work ([2] and under review) lays foundation for a formal specification framework that can support rigorous analysis and also discusses issues regarding formalizing UML. This paper focusses on analysis of designs. The formal specification developed enables us to develop an analysis tool that performs analysis of an object-oriented design irrespective of its specification technique. The Goal Expression Language (GEL) was developed to allow the user to specify analysis goals for the design. This paper provides the abstract syntax of GEL that forms the basis for formal semantics of GEL and demonstrates its use with the help of examples.

The paper is organized as follows. Section 2 discusses some of the issues associated with object-oriented analysis. Section 3 explains the basic components of GEL along with examples. Section 4 discusses the refined syntax of GEL. It explains how the different basic components gel with each other to express different static and dynamic analysis goals. Section 5 presents some examples that demonstrate the use of GEL constructs. It shows how the basic components can be combined together to specify goals. Section 6 summarizes this work and suggests directions for future work.

2. ISSUES AND OBJECTIVES

UML is an excellent design tool that helps in documenting the structure and behavior of a software system. The pictorial notation is extremely intuitive and helps in visualizing and communicating the design decisions to other team members and clients. It is extremely popular because it is highly intuitive, but has a major drawback that it lacks formality. The constraints are expressed using Object Constraint Language (OCL). The semantics of this language are yet to be completely precisely defined [9]. Due to the lack of formality and precision in UML, analysis tools based on UML cannot support rigorous analysis of a design. Recent work in [6] provides formal semantics of UML sequence diagrams. Our formal specification framework converts UML structural design into the formal specification. In this way, the user gets the advantage of using the highly intuitive UML as front-end, and the automatic conversion into formal specification allows for more rigorous analysis. Thus, we have incorporated the advantages of both formal and informal specification techniques.

A number of analysis techniques that were developed, detected inconsistencies among different diagrams ([3], [4], [5]). The main idea behind these techniques is that different views
of the same system should have the same information or more refined views of the same model should also have the same information. Any inconsistencies in these diagrams indicate design errors. These techniques are definitely good at detecting these inconsistencies, but it is not possible to detect subtle design flaws using such techniques. pUML is another framework that performs analysis of a design by proposing a conjecture and then proving the conjecture true using some transformation rules. The basis of this technique is still conjecture, so it might hide some design flaws. Other testing tools were developed to test the behavior of the system ([7], [8]). These are rather involved techniques and we believe that more comprehensive results can be obtained by using simpler techniques. In [1] an analysis tool was developed that performed automated analysis of the design. Automated verification is desirable in many cases, but the kind of analysis it does is more general and applies to a broad range of designs. 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.

We want to be able to provide more customized analysis of the design. Anyone with the knowledge of system requirements and design principles should be able to analyze the design. To analyze the design for finer details that are specific to a particular design, we provide a language that enables the user/analyst to specify his/her analysis goals. This paper addresses these issues and provides a language, Goal Expression Language (GEL) for the analysis of the design. The goals for rigorous analysis of an object-oriented design are as follows:

- The analysis properties should not be hard coded in the tool.
- The analysis of a design should be specific to each design and not just cover a wide range of properties that apply to a number of designs in general.
- Analysis should be based on the analyzer’s requirements/goals.
- Expressing the goals should be a simple task for the user without having to spend too much time in learning new tools or languages.
- At the same time there should be a formal basis for the tool so that it is precise and accurate and there is no confusion about its use.
- The tool should be independent of the design notation used.
- If the design notation evolves or changes completely, the analysis tool should still remain useful.

GEL was designed keeping these goals in mind. The features of GEL are:

- This language can be used to express goals for the static behavior analysis and dynamic behavior analysis of the design.
- It is a very simple language, close to natural language. So, the learning factor is extremely low.
- This language is based on predicates and uses terms and keywords that the designer/analyst is already familiar with.

- GEL will be completely formally defined using denotational semantics, so there is no doubt about its intended use and the correctness of the goals specified.
- GEL is based on the formal specification discussed earlier; hence it has the formal basis that is required to perform rigorous analysis of OO designs. At the same time the designer need not know the details of the formal specification.
- GEL is supported by a processor that performs static behavior analysis of the design based on the goals expressed using GEL.

We refer to static behavior goals as the ones that can be used to analyze the design for the static properties and the design principles of the system, while the dynamic behavior goals are the ones that are used to analyze the actual behavior of the system.

3. ABSTRACT SYNTAX

This section presents the abstract syntax of GEL. It discusses the basic components of GEL using examples. The main idea behind using GEL is to identify the components that need to be analyzed and then specify the goals for these components.

3.1 Set Membership Component

To express analysis goals for different elements of an OOD it is important to first identify the elements themselves. The set membership component helps in doing that. It identifies the instances to which the goals are to be applied.

```
setMembership: designElement(instance)+
```

Laws:

1. The `designElement` can be one of the following keywords: global, package, class, attribute, method, parameter, relationship, rolename, dependency, behavior, event or sequence.
2. The `instance` can be either an identifier, indicating that the instance is determined at runtime by the processor or it can be the name of the `designElement` in that design. For example `class(Student)` indicates an instance of a class determined at runtime, while `class(Student)` specifies a particular class from the design. The first option can be used for analyzing static behavior of the system, while the second one can be used for analyzing the dynamic behavior of the system.
3. If more than one `instance` is used, all the instances should be identifiers or all of them should be specific elements of the design.

3.2 Container Component

Since the elements of an object-oriented design belong to other elements, the container component should be specified along with the `setMembership` component. This is mainly useful when analyzing the dynamic behavior of the system.

```
containerComponent: IN intype (setMembership)+
```

The significance of this component can be realized when using specific instances in the set membership component. For
example when specifying attribute (studentId) it is important to specify the container component IN class(Student).

Laws:

1. More than one setMembership components can be specified at a time.

2. Intype is optional and if used can be either EACH or SAME. This is used only if the instances under consideration have some relationship with previously identified instances or the same goals need to be applied to all the container elements. For example if a certain goal has to be applied to private attributes of each class in a package it can be stated as attribute(a) in each class(c). Similarly, if a goal has to be applied to an attribute of the same class that a previously identified method belongs to then this can be stated as method(m) in class(c); \exists attribute(a) in same class(c).

3.3 Identifying Component

This is the simplest non-iterative form of statement for identifying instances to apply the constraints to.

identifyingComponent: (setMembership) + (containerComponent)

Laws:

1. Multiple setMembership components can be used with a single setMembership component in the containerComponent. For example method(m1, m2), attribute(a) in class(c). In this case both method and attribute belong to class.

2. Multiple setMembership components can also be combined with multiple setMembership components in the containerComponent. In such a case these elements have to be ordered. For example relationship(r), attribute(a) in package(p), class(c). In this example relationship belongs to a package, while attribute belongs to a class.

3.4 Association Component

This component can be used for navigating the design through association relationships using role names.

associationComponent: identifyingComponent

Laws:

1. This uses the identifying component with the restrictions that all instances in the setMembership component cannot be variables. They have to be names of specific elements in the design viz., class names and role names.

2. The navigation can be done by specifying the different associations using their role names in an orderly fashion in the setMembership component.

3. The containerComponent in this case will always be package, since the association relationship and the classes that they work with are contained in a package.

4. In the setMembership component each pair of the components should have an association relationship between them.

Consider the class diagram for package SchoolSystem as shown in figure 1

- The goal each department can enroll at most 50 students can be expressed like this: class(dept), rolename(enrolls) in package(SchoolSystem)
  count(studentId) <= 50

- The goal each student registers for course 101 can be expressed like this: class(Student), rolename(requests), rolename(refersTo) in package (SchoolSystem)
  value(courseId) == 101

3.5 Iterator Component

Iterations can be used to specify the analysis goals on more than one instance. This component can be combined with the conditional operator to specify goals for specific instances.

iteratorComponent: EACH identifyingComponent

In this case the specified goals are expressed and validated for each identified instance. For example each class(c) in each package(p) Stmt1. In this case Stmt1 is executed on each class in each package in the system.

3.6 Universal Component

This component can be used to identify more than one instance and validate the expressed goals for all the elements as one set, as opposed to the iterator component that applies the goals to each instance individually.

universalComponent: ALL identifyingComponent

This component is very useful to verify certain predicates that apply to more than one instance like cycles, count, sum, etc. Consider an example all attribute(a) in each class(c) count(a) > 0. The predicate count is applied to the set of attributes a as opposed to each attribute a.

3.7 Conditional Component

This component is used as a filter for selecting or rejecting instances of a design element. This component is defined by specifying a selection or rejection condition.

conditionalComponent: SUCHTHAT \tau | EXCEPT \tau

The conditional component can be used along with any of the element identifying components so that instances satisfying condition \tau are either selected or rejected.

Laws:

1. The result of evaluating \tau should be Boolean.

3.8 Expression Component

This component specifies the comparison expressions for the identified elements.

comparisonExpression: lexpr op rexpr

Laws:

1. The operator op can be any one of the relational operators \text{==}, \text{!=}, \text{<=}, \text{>=}, \text{<}, \text{>} or the set operators \in, \notin.

2. Also logical operators \land and \lor can be used to combine more than one comparison expressions.
3. The lexpr can be either an instance or a predicate applied to an instance, like name(c), value(c), type(a), etc.

4. Predicates can be nested, e.g. value (first(FName)).

5. The nested predicates are evaluated from inside to out.

6. The rexpr can be either an instance or a predicate applied to an instance or constant values or predefined keywords or arithmetic expressions \( +, -, \times, \div, \% \) or set operations like \( \cup, \cap \) or \( \sim \).

7. The constant values can be integer, real, boolean or string.

8. The result of evaluating a comparisonExpression is a boolean value.

The set of predicates contains the following predefined elements:

\[
\text{predicates} = \{ \text{name(c), type(c), visibility, superclass, subclass, cardinality1, cardinality2, count, number, cycles, value, prevalue, postvalue, some, first, last, next, sum, return} \}
\]

- The predicate count returns the number of elements in a set.
- The predicate sum returns the sum of all the elements in a set. This requires that the elements of the set be either integers or real.
- The predicate number is used as an index to specify an element at a particular position.
- The predicate some replicates the behavior of an existential quantifier.
- The predicate value is used to specify the value of an instance.

- The predicates prevalue and postvalue are used with instances of methods and events to determine the values before and after the invocation of the method/event respectively.
- The keyword return can be used to indicate the return value of a method/event.
- The predicate cycles specifies cycles in a set of elements. These elements can be package, relationship or sequence.

The set of predefined keywords consists of the following elements:

\[
\text{keywords} = \{ \text{public, private, protected, outgoing, incoming, general, primitive, userdefined, aggregation, composition, association, generalization, integer, float, char, bool, string, void} \}
\]

3.9 Negate Component

This component can be used to specify an independent non-comparison expression.

\[
\text{negateComponent: } ! (\text{lexpr})
\]

This component is best used for specifying negation of a predicate on instances in a non-comparison expression. In comparison expressions appropriate relational operators can be used. For example \( ! (\text{cycles(r)}) \). This specifies a goal that no cycles can exist in \( r \).

3.10 Guard Component

This component can be used to specify goals only if certain conditions are satisfied.

\[
\text{guardComponent: } IF(\text{comparisonExpression}) + THEN \tau
\]

Laws:
1. More than one `comparisonExpression` can be specified as the guard conditions. These multiple `comparisonExpression` are combined using logical operators.

2. Only if the `comparisonExpression` are true or are satisfied included statements are executed.

### 3.11 Existential Component

This component can be used to search for at least one element that satisfies a specified condition.

existentialComponent: `THEREIS identifyingComponent
conditionalComponent τ`

**Laws:**

1. Conditional component `SUCHTHAT` is used here.

Consider example `thereis method(m) in class(c)` such that `type(m) = void`. This goal looks for a method in some class that has type void.

### 4. Refined Syntax

This section describes the refined syntax of GEL. It explains how the different basic components of GEL can be combined together.

#### 4.1 Member Component

The member component is used for identifying instances on which the goals have to be applied and it can be any one of the previously defined association, iterator or universal components.

memberComponent: `associationComponent | iteratorComponent | universalComponent`

If a conditional component is specified then set members satisfying those conditions will be selected. The rest will be ignored.

#### 4.2 Object Component

This component can combine the `memberComponent` with the `conditionalComponent` to identify specific elements in a design.

objectComponent: `memberComponent
conditionalComponent | memberComponent`

#### 4.3 Constraint Component

This component is used to specify goals that the identified objects must meet for the analysis to be successful. This component is a set of constraints where each constraint can be one of the options specified below.

constraintComponent :: `constraintSet
constraint : comparisonExpression | negateComponent |
  guardComponent | existentialComponent`

This is the smallest unit of the goal. Whether a single object component is used or a nested component is used, they have to terminate with at least one constraint component. These specify the goals that need to be satisfied.

**Laws:**

1. The different constraints have to be combined using logical operators `∧` and `∨` or can be nested.

Consider example `name(x) == class ∨ name(x) == method ∧
if(name(x) == class) then thereis attribute(y) in class(x)
such that Stmt1`. This example combines three constraints using logical operators and the third constraint has a nested `thereis` constraint in it.

#### 4.4 Goal Design Component

The goal design component uses all the previously defined basic components to specify the entire goal for analyzing a design.

```
goalDesign: memberComponent τ | memberComponent
c conditionalComponent τ
  t: objectComponent τ | objectComponent |
  constraintComponent
```

This definition basically says that the member component with or without the conditional component can be nested with the constraint component to specify analysis goals for a design.

### 5. Examples

This section discusses different examples using GEL. It basically shows how different constructs of GEL can be combined together to specify analysis goals. These examples are simple so that the emphasis is on demonstrating the use of different constructs and following the laws of their use.

1. each class(c1, c2) in each package(p)
   - name(c1) != name(c2)
2. attribute(StudentID), method(AddStudent) in class (Student)
   - type(StudentID) == integer
3. each attribute(a), relationship(r) in class(c), package(p)
   - visibility(a) == private ∧ name(r) != generalization
4. each method(m) in each class(c)
   - thereis attribute(a) in same class(c) suchthat
     name(some(parameter(m)))) == name(a)

These examples explain the use of identifying component while following all the laws of use. Example 1 shows how more than one instances can be used in setMembership component. Example 2 shows the use of more than one setMembership components with a single containerComponent. Example 3 shows the use of multiple setMembership components with multiple containerComponents, while example 4 demonstrates the use of keyword SAME in the containerComponent and also the use of nested predicates. Consider the following examples with reference to figure 1.

5. class(Department), rolename(enrolls) in package (SchoolSystem)
   - count(StudentID) > 0 ∧
     thereis attribute(FName, LName) in class(Student) suchthat
     value(FName) == sonal ∧ value(LName) == dekhane

Example 5 demonstrates the use of association component along with multiple constraints.

6. all relationship(r) in each package(p) suchthat
   - name(r) == generalization !cycles(r))

Example 6 demonstrates the use of `universalComponent` combined with `conditionalComponent` and the `negateExpression`. 

57
behavior analysis and a code generator that generates the
pnu is supported by a processor that performs the static
for any declaration of instances or assignment expressions. The
goal is supported by a processor that performs the static
behavior analysis goals. The dynamic behavior goals should be handled by the code generator to
insert code snippets at appropriate places in the code. This has not yet been implemented. The code generator at this point generates skeletal C++ code based on the design specifications. Considering figure 1 a fragment of the C++ code generated is shown here.

```c++
class Department{
private:
  int deptId;
  string deptName;
public:
  Department();
  ~Department();
  bool AddDept(int Did, string DName);
  bool DelDept(int DId); 
}
```

6. CONCLUSIONS AND FUTURE WORK

This paper has presented an abstract representation of the analysis goal expression language GEL. This is the first step in defining the formal specification of the language. The next step is to specify formal semantics, which will be done using denotational semantics. This paper has also provided examples on the use of GEL. GEL can be used by designers to express their goals for the analysis of the design. It is a very simple language, similar to natural language and can be used to specify static behavior analysis and dynamic behavior analysis goals. It does not change the values of any of the instances being considered and also does not allow for any declaration of instances or assignment expressions. GEL is supported by a processor that performs the static behavior analysis and a code generator that generates the code on successful analysis. The processor implemented includes a parser, a semantic checker that checks if the laws discussed earlier are followed and an analyzer that uses the design information from the specification framework to analyze the static behavior goals of the system. The dynamic behavior goals should be handled by the code generator to insert code snippets at appropriate places in the code. This has not yet been implemented. The code generator at this point generates skeletal C++ code based on the design specifications. Considering figure 1 a fragment of the C++ code generated is shown here.

```c++
class Department{
private:
  int deptId;
  string deptName;
public:
  Department();
  ~Department();
  bool AddDept(int Did, string DName);
  bool DelDept(int DId); 
}
```