Chapter 5

Expressing and Validating OCL Constraints using Graphs

Najet Zoubeir
Institut Supérieur d’Informatique, Tunisia

Adel Khalfallah
Institut Supérieur d’Informatique, Tunisia

Samir Benahmed
Faculté des Sciences de Tunis, Tunisia

ABSTRACT

The definition of the semantics of visual languages, in particular Unified Modeling Language (UML) diagrams, using graph formalism has known a wide success, since graphs fit the multi-dimensional nature of this kind of language. However, constraints written in Object Constraint Language (OCL) and defined on these models are still not well integrated within this graph-based semantics. In this chapter, the authors propose an integrated semantics of OCL constraints within class diagrams, using graph transformation systems. Their contribution is divided into two parts. In the first part, they introduce graph constraint patterns, as the translation into graphs of a subset of OCL expressions. These patterns are validated with experimental examples using the GROOVE toolset. In the second part, the authors define the relation between OCL and UML models within their graph transformation system.

1. INTRODUCTION

The formal definition of the semantics of visual languages has been the focus of many works, in order to extend the scope of such languages to more critical domains. For example, UML diagrams have been formalized using different formalisms, such as formal specification languages (PVS [Aredo, 1999; Ledang & Souquiers, 2001], CSP [Ng & Butler, 2003], Z [Dupuy, 2000; France & Bruel, 2001; France, Bruel, Larrondo-Petrie, & Grant, 1997]...). Among these formalisms, graph transformation systems have known a fair success representing the visual languages seman-
Expressing and Validating OCL Constraints using Graphs

tics, since this formalism is formal, universal and easily understood. Many graph-based semantics were proposed for UML diagrams, such as the Dynamic Meta-Modeling approach introduced by Hausmann, which formalizes many of the UML diagrams, such as the statechart diagrams (Engels, Hausmann, Heckel, & Sauer, 2000; Hausmann, 2001), the sequence diagrams (Hausmann, Heckel, & Sauer, 2002; Hausmann, Heckel, & Sauer, 2004) and the activity diagrams (Hausmann, 2005). Other works try rather to define integrated semantics for a number of UML diagrams. For example, (Kuske, Gogolla, Kollmann, & Kreowski, 2002), and (Gogolla, Ziemann, & Skuske, 2003) propose a graph-based integrated semantics for UML class, object and statechart diagrams, and (Holscher, Ziemann, & Gogolla, 2006) works on a larger subset of UML diagrams, including further the use cases and interaction diagrams.

In this context, expressing constraints on UML diagrams, written in the Object Constraint Language (OCL) is studied in many works (Bauer, 2008; Rutle, Rossini, Lamo, & Wolter, 2012; Dang & Gogolla, 2009; Rensink & Kleppe, 2008; Bottoni, Koch, Parisi-Presicci, & Taentzer, 2002). In general, the purpose of these works is to provide a semantics of OCL constraints using graphs. The work in (Bauer, 2008) defines the notion of conditions on attributes in DMM graphs, without considering the OCL syntax, since he does not directly manipulate UML diagrams. In fact, the author considers conditions on graph nodes as an additional refinement of the matching in graph transformation rules. He uses the GROOVE toolset (Groove, 2012) for the representation and manipulation of graphs and constraints. Rutle et al. in (Rutle, Rossini, Lama, & Wolter, 2012) propose constraint aware model transformations based on the diagram predicate framework (DPF), which is a generic graph-based specification framework. The authors propose to define constraints in the transformation rules specified by a joined modeling language, used to join the source modeling language to the target modeling language. The constraints are written in First-Order Logic (FOL), and represented in diagrams by diagrammatic signatures. Although this approach is based on the formal meta-modeling framework DPF, we consider that the proposed constraint semantics (1) is not well integrated with models, since it uses a different notation (FOL), and (2) is restricted to the meta-level, because constraints are used only to control which structure to create in the target model, and which constraints to add to the created structure.

The work of Dang et al. in (Dang & Gogolla, 2009) propose to integrate OCL constraints with Triple Graph Grammars (TGG), and realize this approach by a tool as an extension of USE (UML based Specification Environment). OCL conditions are used in meta-models in order to precisely represent well-formed models, and in models as their properties. However, OCL constraints are represented in the proposed approach in their textual form. So the integration of OCL with TGG is carried out by the tool, and not in a visual form. The work presented in (Resink & Kleppe, 2008) formally extends type graphs to be more compliant with UML. This extension includes a set of object oriented notions such as inheritance and multiplicity, and defines the notion of graph constraints. The work organizes these constraints in categories, such as bidirectionality and multiplicities on associations, and acyclicity of containments, and proposes formal definition for each category. However, the authors do not specify the details of OCL constraints since they do not aim to define a semantics for them, and just classify them as a general category. Bottoni et al. in (Bottoni, Koch, Parisi-Presicci, & Taentzer, 2002) propose an OCL constraints visualization based on collaboration diagrams, and an operational semantics based on graph transformation units. They propose to express constraints by graph transformation rules, such as a constraint satisfaction is represented by the matching between the rule and the instance graph. However, the OCL constraints are manipulated as expressions, without defining how
these expressions are integrated within models. In fact, the authors use OCL invariants in their examples, but give no details about the use of the other OCL placements, such as preconditions and postconditions.

Considering the set of works previously presented, we noticed that in the definition of UML diagrams semantics using graph transformations, the semantics of OCL constraints is whether defined in a non-accurate way (Resink & Kleppe, 2008), or defined separately from UML diagrams and hence lacks of a solid integration in the graph transformation systems (Rutle, Rossini, Lamo, & Wolter, 2012; Dang & Gogolla, 2009; Bottoni, Koch, Parisi-Presicci, & Taentzer, 2002). In this chapter, we propose an integrated semantics of OCL constraints within class and object diagrams, using graph transformation systems. We propose to represent models composed by class and object diagrams and OCL constraints in an integrated form using graph transformation systems, as depicted in the Figure 1.

Our main contribution is the proposition of an integrated semantics of OCL constraints based on graph transformation systems. In a first part, we propose a number of patterns representing a set of OCL expressions using graph notations. These patterns allow translating OCL constraints to graph constraints. We relied on experimental examples using the GROOVE toolset in order to validate our patterns. Then, in a second part, we explain how to use and interprete these graph constraints within graph transformation systems, in the same way as OCL constraints are used and interpreted within a UML model.

The remainder of this paper is organized as follows: In section 2 we will briefly present the notions of graphs and graph transformations. In section 3 we will illustrate the graph constraint patterns we propose, and section 4 will describe the use of these patterns in graphs representing class diagrams, in a way similar to the use of OCL constraints in UML models. Finally, in section 5 we will exemplify our proposal using an illustration example.

2. CONSIDERED GRAPHS

In our work, we consider the graphs defined in (Hausmann, 2005): “The graph notion (...) is a typed, attributed, node and edge labeled multi-graph that allows for node inheritance in the type graph and uses special datatype nodes (DT) for the representation of attributes.” So we deal with type graphs, instance graphs and rule graphs, such as the operation semantics is described by transformation rules. The following definitions are used to present the different concepts belonging to a graph transformation system (Hausmann, 2005). In these definitions, we consider a combined label alphabet $\Lambda$ providing labels to nodes and edges, and containing the special label $\perp$ as defined in (Hausmann, 2005). We consider also an extension of this alphabet called $\Lambda' = \Lambda \cup \{\bullet\}$ which contains the wildcard symbol $\bullet$ which evaluates to true when compared to any other label of $\Lambda'$ (Hausmann, 2005).

**Definition 1 (Graph):** $G = (N, E, l_N)$ with

- $N$ the finite and non-empty set of nodes,
- $E$ the finite set of edges,
- $l_N$ the labeling function from $N$ to $\Lambda'$.

![Figure 1. The integration of class and object diagrams with OCL constraints in graph transformation systems](image-url)
Expressing and Validating OCL Constraints using Graphs

- The finite set of directed edges, \( E \subseteq \mathcal{N} \times \mathcal{A} \times \mathcal{A} \times \mathcal{A} \times \mathcal{N} \),
- \( l_N : \mathcal{N} \to \Lambda \) the labeling function for nodes.

**Definition 2 (Edge-Label Preserving Graph-Morphism (elp-morphism)):** An elp graph morphism is a structure and edge label preserving morphism between graphs.

\[
m(\mathcal{G}, \mathcal{H}) = m_N \text{ with } m_N : \mathcal{N}^G \to \mathcal{N}^H \text{ the node mapping function, and } \forall a, b, c, d \in E^G : m_N(a, l_a, l_b), m_N(b, l_b, l_c), m_N(c, l_c, l_d) \subseteq E^H
\]

**Definition 3 (Typed Graph):** \( \mathcal{G}_T = \mathcal{G}, \text{type} \) with \( \mathcal{G} \) a graph as defined above, \( \text{type} : \mathcal{G} \to \mathcal{T} \) the typing elp-morphism and \( \mathcal{T} \) a graph with the additional requirement that \( l^G_N \) is injective.

**Definition 4 (Type Graph with Inheritance):** Inheritance in the type graph \( \mathcal{T} \) is expressed by special edges \( I \). \( \mathcal{G} = \mathcal{T}, I, A \) with \( \mathcal{T} \) a graph with the additional requirement that \( l^I_N \) is injective \( I \subseteq \mathcal{N}^G \times \mathcal{N}^T \) the set of inheritance edges which must not form a circle \( A \subseteq \mathcal{N}^T \) the set of abstract nodes.

We consider graphs that were designed by (Hausmann, 2005) as DMM Type Graphs, DMM Instance Graphs and DMM Rule Graph.

**Definition 5 (DMM Type Graph):**

\[
\mathcal{G}_{\text{DMT}} = \{ \mathcal{N}, \mathcal{E}, l_N, I, A, DT \} \text{ with } \mathcal{N}, \mathcal{E}, l_N, I, A \text{ a type graph with inheritance } DT \subseteq \mathcal{N} \text{ the set of data types } l_N(\mathcal{N}) \subseteq \Lambda \setminus \{ \bot \}
\]

\[
E \subseteq \left( \mathcal{N} \setminus DT \right) \times \Lambda \times \Lambda \times \Lambda \times \left( N \setminus DT \right)
\]

\[
\bigcup \left( \mathcal{N} \setminus DT \right) \times \{ \bot \} \times \{ \bot \} \times \{ \bot \} \times \Lambda \times DT \big).
\]

**Definition 6 (DMM Instance Graph):**

\[
\mathcal{G}_{\text{DIM}} = \{ \mathcal{G}, \text{type with } \text{type} : \mathcal{G} \to \mathcal{T} \} \text{ the typing elp-morphism}
\]

\[
TG \subseteq \mathcal{G}_{\text{DMT}}
\]

\[
l_N(N^G) \subseteq \Lambda
\]

\[
l_k(E^G) \subseteq \Lambda \times \Lambda \times \Lambda
\]

\[
\forall s \in DT^T \text{ : type} \subseteq \{ s \} \subseteq D_s \}
\]

\[
TG \text{ is the Concrete Transitive Closure defined in (Hausmann, 2005).}
\]

**Definition 7 (DMM Rule Graph):**

\[
\mathcal{G}_{\text{DMR}} = \{ \mathcal{G}, \text{type with } \text{type} : \mathcal{G} \to \mathcal{T} \}
\]

\[
TG \subseteq \mathcal{G}_{\text{DMT}}
\]

\[
l_N(N^G) \subseteq \Lambda
\]

\[
l_k(E^G) \subseteq \Lambda \times \Lambda \times \Lambda
\]

\[
\forall s \in DT^T \text{ : type} \subseteq \{ s \} \subseteq D_s \}
\]

\[
TG \text{ is the Abstract Transitive Closure defined in (Hausmann, 2005).}
\]

According to these definitions, we refine Figure 1, such as a class diagram will be translated to a type graph; an object diagram to an instance graph, and the OCL expressions to a set of graph transformation rules as depicted in Figure 2. Our contribution is composed by two main parts: First we define a set of graph constraint patterns representing the translation into graphs of OCL expressions. The definition of these patterns is based on experimental examples, and validated using the Computation Tree Logic (CTL) within the toolset GROOVE. In the second part of our
Expressing and Validating OCL Constraints using Graphs

In our work, we are trying to represent OCL constraints within graphs representing class diagrams. In order to do that, we propose a set of graph constraints patterns. These patterns was created according to a subset of the OCL expressions abstract syntax (illustrated by the meta-model of Figure 3) and concrete syntax, both described in (OMG, n.d.). We note that we do not consider constraints figuring as adornment in UML diagrams, such as \{ordered\} and \{xor\}, but only OCL constraints defined by the Object Management Group (OMG) in (OMG, n.d.).

In order to validate our patterns, we relied on experimental examples of expressions that we verified in some cases, using CTL within the GROOVE toolset. CTL is used with the invariant constraints to check that the graph constraint expresses the exact meaning of the OCL constraint, in the full state space of the corresponding transition system generated by GROOVE (more details in section 4.1). The patterns presented in this paper correspond to the following meta-classes from the OCL expressions meta-model: LiteralExp, VariableExp, and FeatureCallExp. Considering the facts that the meta-classes MessageExp and StateExp concern the UML statechart diagram, and the meta-class LoopExp (and its descendants) is a repetitive form of an OCL expression, we did not propose corresponding patterns for them. The use of a simple-graph transformation tool such as GROOVE to manipulate multi-graphs should not be a problem, since it is possible to translate typed multi-graph production systems into typed simple-graph production systems (Boneva, Hermann, Kastenberg, & Rensink, 2007).

In what follows, we will present our graph constraints patterns, using illustrations based on GROOVE notations, and eventually accompanied by descriptions and examples. In each pattern visualization, the red-colored elements are generic elements, that have to be replaced with specific values or graph elements when instantiating the pattern.

3.1. Literal Expression Pattern

A literal expression of a type T is a value of this type. Using the GROOVE notation, this is represented by a node labeled with the type followed by the literal value, as depicted in Figure 4. We

Figure 2. Refinement of the integration of class and object diagrams with OCL constraints in graph transformation systems
should mention that we do not deal with NULL literal and invalid literal expressions. This pattern is used for constants and valued attributes.

### 3.2. Variable Expression Pattern

In OCL, a variable expression is just a name that refers to a variable or self (OMG, n.d.). In our context, the keyword self will be represented by a node belonging to the instance graph, to which a graph constraint expression is linked, as it will be explained in section 3.3. So the Variable Expression Pattern is used to represent variables, as illustrated in Figure 5(a).

### 3.3. Feature Call Expression Patterns

In this category we propose two patterns: the Property Call Expression Pattern and the Operation Call Expression Pattern.

#### 3.3.1. Property Call Expression Pattern

A Property Call Expression in OCL refers to an attribute of a class, and evaluates to the value of this attribute (OMG, n.d.). In the concrete syntax of this expression, the property name is marked with the keyword ‘@pre’, which refers to the value of a feature at the start of the operation. This value will eventually be modified by the operation. In graphs, the use of ‘@pre’ is represented by a node representing an attribute, that will be deleted, in order to be replaced by a new one containing the new value. The Property Call Expression Pattern generic structure is depicted in Figure 5(b), such as the blue color is used to represent a node that will be deleted by a graph transformation rule.

#### 3.3.2. Operation Call Expression Patterns

This category contains four patterns: the Operators Pattern, the Universal Quantifier Pattern, the
3.3.2.1. Unary and Binary Operator Pattern

In order to represent operators, we use the notion of product nodes defined in GROOVE (Rensink, Boneva, Kastenberg, & Staijen, 2011). Product nodes are related to their operands (which are expressions) by Argument Edges. These nodes are generally used in the instance level. We will use it in the pattern visualization in a generic form with expression patterns to express Unary and binary operator Patterns. Figure 6 illustrates the Binary Operators pattern: the node \( P \) is the operator and the \( \pi_0 \) and \( \pi_1 \) edges link the operator respectively to its first and second operands, which are expressions.

The result will be visualized by a node presenting its type and value, linked to the product node with an edge labeled by the operation carried out on the arguments. The Unary Operator Pattern is similar to the Binary Operator Pattern, with only one argument \( \pi_0 \).

3.3.2.2. Universal and Existential Quantifier Pattern

The Universal Quantifier Pattern is used to illustrate OCL expressions with the keyword forAll. In order to represent these expressions with graphs, we interpreted the expression \( \text{Collection}.\text{forAll} (\text{expression}) \) as for all the elements in \( \text{Collection} \), there exists \( \text{expression} \). In GROOVE notation, we need to use an existential node nested in a universal node in order to express OCL expressions with the keyword forAll.

The OCL expressions with the keywords exists and includes are boolean expressions expressing whether a certain expression is or not satisfied by at least one element of the collection. In GROOVE notation, the existential node cannot be used by its own, it should be always nested in a universal node.

That is why we define a pattern for both expressions. This pattern is depicted in Figure 7. The node of type ”Class” linked to the universal.
node represents the objects of the collection, and all the nodes belonging to the expression must be linked to the existential node.

4. THE USE OF GRAPH CONSTRAINTS IN GRAPHS

In the definition of the use of OCL in UML models, described in the OCL specification document (OMG, n.d.), the authors distinguish three things that need to be separated for every OCL expression occurrence: the placement, the contextual classifier, and the self instance.

4.1. Placement

The placement represents the position where the OCL expression is used in the UML model. The standard placements of OCL expressions within class diagrams are summarized by (Yang, 2004) as shown in the Figure 8.

In the Figure 8, the placements are association-ends roles associated to the class oclExpression that belong to the OCL meta-model. We notice that the author did not represent the class Constraint from the UML meta-model, whose role is to relate the expressions to their constrained elements. Except for the initial value of an attribute and the definition expressions, which we did not consider in this paper, all the graph constraints placements have to be expressed in graph transformation rules as explained below:

1. **Pre-Condition**: A pre-condition is a boolean expression associated to an operation. It defines a condition that must hold before the execution of that operation. Since an operation semantics is depicted by a graph transformation rule in our proposal, a pre-condition associated to this operation will be represented by a graph expression linked to one or more nodes in the graph transformation rule expressing the operation semantics.

2. **Post-Condition**: A post-condition is also a boolean expression associated to an operation, that corresponds to a condition that must hold after the operation executes. In the context of graphs, a post-condition can be expressed only if it concerns a new created node. The creation of nodes in a graph transformation rule corresponds to whether a modification of an attribute’s value, or to the creation of a new object. So a post-condition will be represented in the graph transformation rule associated to the considered operation, by a graph expression linked to one or more new created nodes.

3. **Operation Body**: This type of expressions concerns query operations. It is simply represented by the graph transformation rule expressing the query operation semantics.

4. **Invariant**: By definition, an invariant is a boolean expression over a class (classifier in general) that must be verified by all the instances of this class. In the context of graphs, in order to define invariants, we have to ensure that all the nodes representing a class instances verify the considered invariant expression during the whole simulation, which includes also the new nodes created by transformation rules.

We propose to represent a graph constraint invariant with an empty side-effect graph transformation rule, which contains the
invariant expression, according to the graph constraint patterns already presented. This rule must be verified in the whole state space during the simulation. That is why, invariant rules have to be checked using CTL formulas. For example, let I be the following OCL invariant:

\[
\text{context Companie} \\
\text{inv : Companie.employe->} \\
\text{forAll(e:Employe|e.age < 65)}
\]

And let R be the corresponding rule to I. R is depicted in Figure 9. To ensure that R is verified for all instances, we have to check that the CTL expression holds for all states. The result of this formula depends on the host graph.

\[
\text{AG } R
\]  

(1)

5. **Derivation:** By definition, a derived value expression is an invariant that states that the value of the attribute or association end must always be equal to the value obtained from evaluating the expression. That is why

---

**Figure 8. Standard placements in OCL (Yang, 2004)**

**Figure 9. Example of an invariant expression**
4.2. Contextual Classifier

By definition, the contextual classifier defines the namespace in which the expression is evaluated. In the previous figure, the contextual classifiers are the classes imported from the UML meta-model, which are: Classifier, Attribute and Operation. In the translation to the graph formalism, all of these three elements are represented by nodes. So the contextual classifier will be always represented by a node, to which a graph constraint expression is linked.

4.3. Self Instance

As the self instance is always an instance of the contextual classifier, we propose to represent it by a node belonging to the instance graph, to which a graph constraint expression is linked.

5. ILLUSTRATION EXAMPLE

We devote this section to an illustration example, which illustrates how a class diagram accompanied with OCL constraints is translated to a graph transformation system, according to our proposal. Our example is the Circular Buffer, inspired from GROOVE toolset samples. The circular buffer class diagram and OCL constraints are depicted in Figure 10.

In the context of the class “Buffer,” the invariant expresses the fact that the attribute length is...
Expressing and Validating OCL Constraints using Graphs

Figure 12. Graph transformation rule for the “get” operation with its pre and post conditions

Figure 13. Graph transformation rule for the “put” operation with its pre and post conditions
a positive value less than or equal to the buffer size. For the “get” operation, the precondition says that it should be at least one element in the buffer, and the postcondition expresses the fact that the buffer length should be reduced by 1 after a “get” operation. The “put” operation precondition requires that the buffer should not be full, and its postcondition expresses the fact that the buffer length should be increased by 1 after a “put” operation.

In the rest of this section, we will represent our graphs (type graph and rule graphs) using GROOVE.

### 5.1. From a Class Diagram to a Type Graph

Since the circular buffer example is inspired from the GROOVE toolset samples, the type graph corresponding to the class diagram of Figure 10(a) we propose is close to the one proposed by the toolset, with some minor modifications, which are the use of the attributes length for the number of the occupied cells and size for the buffer capacity. Figure 11 illustrates the type graph corresponding to the circular buffer class diagram of Figure 10(a).

### 5.2. Representing OCL Constraints in Graph Transformation Rules

#### 5.2.1. “Get” Operation Constraints

The “get” operation has a precondition and a postcondition (cf. Figure 10b):

- The precondition constraint is an operation call expression, corresponding to a binary operator, which pattern is depicted in Figure 6;
- The postcondition constraint is composed by a property call expression, which is the

---

**Figure 14. Graph transformation rule for invariant inv1**

![Graph transformation rule for invariant inv1](image1)

**Figure 15. Considered instance graph**

![Considered instance graph](image2)
operand of an operation call expression, also corresponding to a binary operator.

According to our proposal, the “get” operation semantics is represented by a graph transformation system. In this rule should be represented the preconditions and the postconditions constraints related to the considered operation. The graph transformation rule corresponding to the “get” operation and its pre and postconditions is depicted in Figure 12.

5.2.2. “Put” Operation Constraints

Similarly to the “get” operation, the “put” operation also has a precondition and a postcondition (cf. Figure 10b), corresponding respectively to a binary operator call expression, and a property call expression, which is the operand of a binary operator call expression. The graph transformation rule corresponding to the “put” operation and its pre and postconditions is depicted in Figure 13.

5.2.3. The Invariant Constraint

The Buffer invariant, named \( inv_1 \) (cf. Figure 10b), is composed by the conjunction of two binary operator call expressions. Following our proposal, we created a graph transformation rule representing the invariant constraint, using multiple instances of the binary operator pattern, as illustrated in Figure 14. Then, we used the GROOVE model checker to verify that this rule holds for all the states, for the instance graph depicted in Figure 15. This means that the invariant graph constraint is verified for all the buffer instances generated by the application of the “get” and “put” transformation rules, applied on the considered instance graph.

6. DISCUSSION AND FURTHER WORK

In this paper, we proposed an integrated semantics of UML class and object diagrams accompanied by OCL constraints. After presenting the basic concepts, we started by the definition of a set of graph constraint patterns, used to express OCL constraints within graphs, representing class diagrams. Our patterns are validated using CTL formulas on experimental examples. Then we clarified the use of these graph constraints within graphs representing class and object diagrams, in accordance with the OCL specification document. Finally, we exemplified our proposal using an illustration example, showing the representation of a class diagram accompanied with OCL constraints by a graph transformation system.

Positioning this work within the framework of MDA four layered architecture (OMG, 2003), type graphs will represent the M1 level, and instance graphs will correspond to the M0 one. However, due to the universal character of graphs, this mapping can be generalized such as type graphs and instance graphs will correspond respectively to \( M_{n+1} \) and \( M_n \) levels, for \( n \in [0..3] \). So we will be able to define and validate OCL constraints on graphs representing models and meta-models. In the future this work, we aim to validate models by studying the constraints preservation when a transformation, such as design patterns introduction, is applied on these models.

REFERENCES


Expressing and Validating OCL Constraints using Graphs


Graph Transformation System (GTS): A GTS is the set of graph transformation rules with eventually type graphs and instance graphs.

GRaphs for Object-Oriented Verification (GROOVE): GROOVE is a graph transformation framework developed at the University of Twente, The Netherlands. It focuses on the use of simple graphs for the design, compilation and run of systems modeled as Graph Transformation Systems.

Model Transformation: It is the process of converting a model to another model within the same system. The input and output models should satisfy source and target meta-models.

Object Constraint Language (OCL): OCL is a formal language standardized by the Object Management Group (OMG) for the expression of constraints in UML models. Originally, this language was an IBM contribution to UML 1.1; it has become a part of the modeling standard since UML 1.3.

Unified Modeling Language (UML): UML is an object-oriented Visual Modeling Language (VML), which combines existing object-oriented VML (Booch, OMT and OOSE) and provides diagrams for modeling different aspects of systems. UML is now the most popular industry standard VML in software engineering.

Visual Language (VL): A VL is defined by its concrete and abstract syntax, besides its semantic domain and mapping. In general, the abstract syntax of VL is captured by meta-modeling.

KEY TERMS AND DEFINITIONS

Computational Tree Logic (CTL): CTL is a branching-time temporal logic, expression state and path formulas on the state space. CTL is a restriction of the full expressive tree logic CTL*.  

Graph Transformation System (GTS): A GTS is the set of graph transformation rules with eventually type graphs and instance graphs.

GRaphs for Object-Oriented Verification (GROOVE): GROOVE is a graph transformation framework developed at the University of Twente, The Netherlands. It focuses on the use of simple graphs for the design, compilation and run of systems modeled as Graph Transformation Systems.

Model Transformation: It is the process of converting a model to another model within the same system. The input and output models should satisfy source and target meta-models.

Object Constraint Language (OCL): OCL is a formal language standardized by the Object Management Group (OMG) for the expression of constraints in UML models. Originally, this language was an IBM contribution to UML 1.1; it has become a part of the modeling standard since UML 1.3.

Unified Modeling Language (UML): UML is an object-oriented Visual Modeling Language (VML), which combines existing object-oriented VML (Booch, OMT and OOSE) and provides diagrams for modeling different aspects of systems. UML is now the most popular industry standard VML in software engineering.

Visual Language (VL): A VL is defined by its concrete and abstract syntax, besides its semantic domain and mapping. In general, the abstract syntax of VL is captured by meta-modeling.