Specification of a Language-Based Editor for UML/OCL documents using a Generator System

by
Amarnath Anumandla

Project Report

International Master Program in Computational Logic
Department of Computer Science
Dresden University of Technology
01062 Dresden, Germany

Supervised by : Dipl.-Inf. Enrico Bormann

Contents

Abstract

The objective of the project is to build a specification of UML/OCL documents. This specification is used in generating a language-based editor with the help of the generator system [Bor04]. The specification is written in Dresden Specification Language (DSL) [Bor04]. The UML/OCL documents are defined by means of context-free syntax and context-sensitive syntax.

1 Introduction

1.1 An Editor

An editor in general is a program designed to perform such editorial functions as rearrangement or modification or deletion of data, designed to handle primarily ASCII files. The most often-used ones in UNIX are vi, and Emacs.

1.2 Language-Based Editors

Language-based editors are editors which are aware of the syntactic rules of a specific programming language and provide a framework for the user to enter source code. Language-based editors can be programmed by hand, or can be generated using a generator system. The generated editor makes sure that the program edited is structurally correct according to the context-free syntax and checks for violations of the context-sensitive restrictions.

The main feature of a language-based editor is its immediate-computation to perform analysis, translation and error reporting while an object is being edited. So each modification causes an immediate effect.

1.3 A Language-Based Editor for UML/OCL documents

This report is about specification of a language-based editor for Unified Modeling Language (UML) and Object Constraint Language (OCL) documents [Obj03]. The documents consist of textual form of UML class-diagrams and the set of OCL constraints, where OCL is used to write constraints on UML class-diagrams. The purpose of the editor is to enable the user to write constraints for UML class-diagrams so that it should satisfy context-free syntax and context-sensitive restrictions.

1.4 Automatic Generation of Language-Based Editors

A generator system takes a specification of a particular formal (programming-) language as input and generates a language-based editor for that specified language. For example, the synthesizer generator [RT89] is a system for constructing language-based editors. The person who specifies a formal (programming-) language as input for a generator system is called specifier, and the person who edits a program in the editor, which is output of a generator system, is called user. A specifier should specify a formal (programming-) language in a certain specification language, such that a generator system can understand.

A specification of a language consists of context-free syntax [ASU86] and context-sensitive syntax [ASU86]. Context-free syntax is a grammar in Extended Backus-Naur Form (EBNF) notation, which is a superset of that language. Context-sensitive syntax is specified as list of rules (or restrictions) of that language to restrict the EBNF rules to the language. A generator system takes this specification as input and generates a language-based editor for manipulating objects according to the given rules.

The generator system in [Bor04] takes a specification of a language as input and generates a language-based editor for that language. It understands the specification language DSL [Bor04] only, so the specification must be an element of DSL.

1.5 DSL

DSL is Dresden Specification Language [Bor04], developed at Dresden University of Technology, Germany. It is based on the functional programming language Haskell [FHP98]. A specifier can specify any formal (programming-) language in DSL. The context-free syntax and the context-sensitive syntax of DSL can be found in [Bor04] (Chapter 5). The purpose of DSL is to specify a formal (programming-) language and provide it to the generator system [Bor04] as an input to generate a language-based editor for that specified language. How to specify a formal (programming-) language in DSL is briefly discussed in future section, and detailed information can be found in [Bor04] (Chapter 6).

1.6 UML

Unified Modeling Language (UML) [Obj03] is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The specification can consists of UML class-diagrams and constraints on these diagrams for well-formed semantics. So UML semantics consists of Object Constraint Language (OCL) specification [Obj03] (Chapter 6), which is used to write constraints on UML class-diagrams.

A sample UML class-diagram is shown in Figure 1, which models a student-seminar-tutor relation and specifies as : A student has a name and attends a course. A tutor has a name and belongs to some group. A tutor offers some seminars which are participated by some students. A seminar has a topic and supervised by one or more tutors. A seminar has a start time, an end time, and the duration of the seminar as an operation. Where the *Student*, the *Tutor* and the *Seminar* are classes, and the *Participated_by* and the *Supervised_by* are associations (i.e relations). An association can have roles. So in this example, *participant* and *participatesIn* are role-names of the association *Participated_by* between the *Student* and the *Seminar* classes. *supervisor* and *offers* are role-names of the association *Supervised_by* between the *Tutor* and the *Seminar* classes.

1.7 OCL

OCL is Object Constraint Language [Obj03] (Chapter 6), a formal language to express side-effect free constraints. It is purely an expression language, since constraints are expressions, it guaranteed to be without side-effect.

OCL is a part of the official Object Management Group (OMG) standard for UML (from version 1.1 on). This project used OCL of OMG-Unified Modeling Language v1.5. Constraints in OCL specify invariant conditions that must hold for the system being modeled and pre- and post- conditions for operations.

A UML diagram, such as a class-diagram, is not refined enough to provide all the relevant aspects of a specification. OCL has been developed to fill this gap. It is easy to read and write. Since it is purely an expression language, it is guaranteed to be side effect-free. When an OCL expression is evaluated, it simply returns a value. It can not change the state of the model. A constraint is formulated on the level of classes, but its semantics is applied on the level of objects. So OCL does not contain any "commands" to change the system state, but just operations to observe the system state.

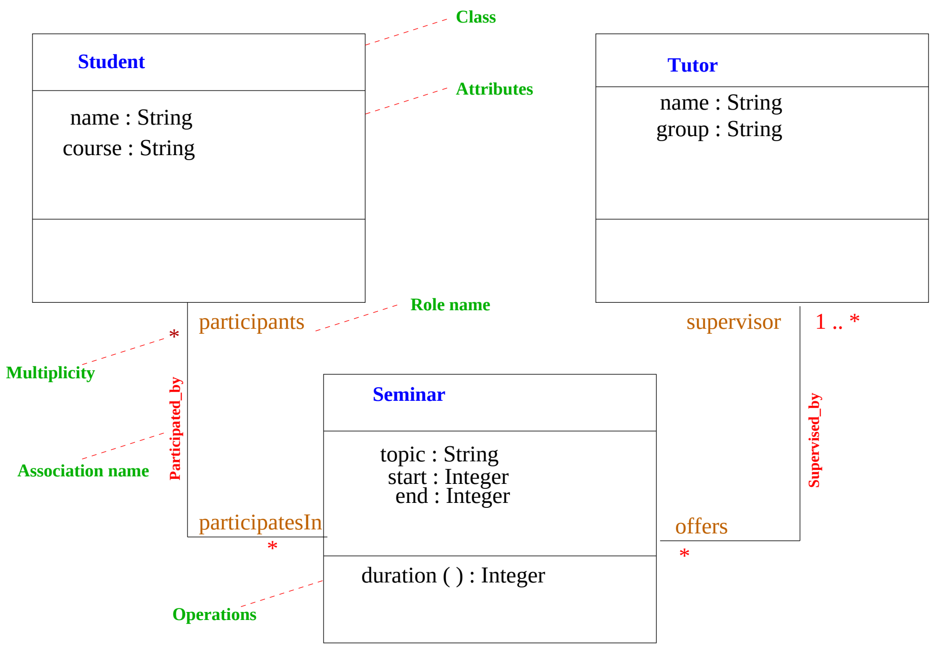

Figure 1: A UML class-diagram on Student-Seminar-Tutor relation

OCL is a typed language, so that each OCL expression has a type. Therefore type conformance is the main restriction of the language. Each classifier defined within a UML model represents a distinct OCL type.

Some constraints on a UML class-diagram of Figure 1 are below.

```
context Seminar
inv : self . end > self . start
```

The above OCL constraint specifies the invariant condition, which says that the end time of the *Seminar* must be greater than the start time of the *Seminar*. The name **self** refers to an instance of the class *Seminar*. It should hold for all objects of the class *Seminar*.

```
context Seminar :: duration ( )
post : result = self . end - self . start
```

The above OCL constraint specifies the post condition on an operation, which says that the result of the **duration()** has to be the difference between the start and the end times. The name **self** is used in referring to the object of the class *Seminar* on which the operation was called.

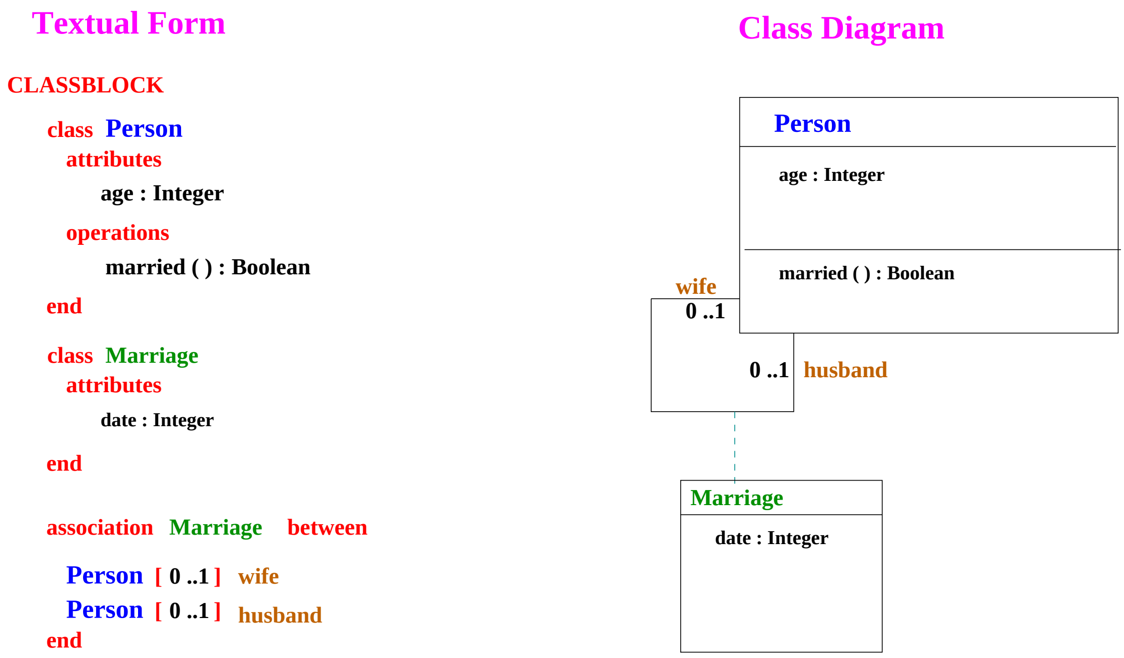

Figure 2: textual representation of a UML class diagram

2 Definition of UML/OCL documents

Here the context-free and the context-sensitive syntax of UML/OCL documents is defined. Since OCL constraints are related to UML class-diagrams, the documents contain the Class Block for textual form of UML class-diagrams and then OCL Block for OCL constraints. The context-free syntax of the documents is defined for Class Block and OCL Block based on OMG-UML-OCL specification v1.5.by means of Extended Backus-Naur Form (EBNF)-definitions. Then context-sensitive restrictions of the UML/OCL documents are defined.

Class Block : The Class Block consists of textual form of UML class-diagrams. A document should consist of one or more class declarations, and zero or more association declarations. Each class declaration has a name, and some set of attributes and operations. Each association has a name and two associated classes. An association is a class with empty attributes and operations. To specify the members (i.e the attributes or the operations) on an association, the user has to declare it as a class with the same name and specify the members. These associated classes can be same or different. If an association is on the same class then it is called recursive association. An associated class has a name, which should be declared as a class in the class declaration, followed by an optional multiplicity, and followed by an optional role name. A role name of an associated class can be viewed as a role of the associated class in that association. If an association is recursive then the role name must be specified.

A sample specification of representation of the textual form of UML class-diagram is shown in Figure 2. In the figure, different colours are used to make it easy to understand.

OCL Block : The OCL Block consists of set of contexts to write constraints on the textual form of UML class-diagrams. Each context consists of a classifier followed by a set of constraints on the classifier. A classifier can be a class alone or with its operation. A constraint can be a stereo type or a definition constraint. The stereo type constraints can be invariant, or pre- , or post- conditions followed by boolean expressions. The definition constraint consists of set of let expressions, which define new attributes or operations of that classifier.

2.1 Context-Free Syntax

The context-free syntax is given by the following rules of an EBNF-definitions, where terminal symbols are enclosed between " and ", | means a choice, ? optionality, * means zero or more times, + means one or more times, −− used for comments, and program is the start symbol.

```
program            := "model" uppName classBlock
                      "constraints" oclBlock  "endmodel"
classBlock         := classDecl+  associationDecls*
classDecl          := "class" uppName
                              attributeDecls*
                              operationDecls*
                      "end"
attributeDecls     := "attributes" attributeDecl+
attributeDecl      := lowname ":"  simpleType
operationDecls     := "operations" operationDecl+
operationDecl      := lowName
                       "(" formalParameterList? ")"
                        ( ":" simpleType )
associationDecl    := "association" uppName "between"
                            assoClassName
                            assoClassName
                      "end"
assoClassName      := uppName ( "[" multiplicity "]" )?
                                ( "role" lowName )?
multiplicity       := mult | mult ".." mult
mult               := ["0"-"9"] ["1"-"9"]*  | "*"

oclBlock           := constraint*
constraint         := contextDeclaration
                      ( ( "def" uppName? ":" letExpression*)
                        |
                        (stereoType lowName? ":" oclExpression)
                      )+
contextDeclaration := "context"
                      ( operationContext | classContext )
classContext       := ( lowName ":" uppName )
                    | uppName
```

```
operationContext   := uppName "::" lowName
                      "(" formalParameterList ")"
                      ( ":" returnType )?
returnType         := type
type               := simpleType | collectionType
collectionType     := collectionKind
                      "(" simpleType | uppName ")"
stereoType         := ( "pre" | "post" | "inv" )
formalParameterList:= ( lowName ":" simpleType
                        ("," lowName ":" simpleType)*
                      )?
oclExpression      := ( letExpression+ "in")? expression
expression         := logicalExpression
letExpression      := "let" lowName
                      ( "(" formalParameterList ")" )?
                      ( ":" type )?
                      "=" expression
ifExpression       := "if" expression
                      "then" expression
                      "else" expression
                      "endif"
logicalExpression  := relationExpression
                      ( logicalOperator
                        relationExpression
                      )*
relationExpression := additiveExpression
                        ( relationalOperator
                          aditiveExpression
                        )?
additiveExpression := multiplExpression
                      ( addOperator
                        multiplExpression
                      )*
multiplExpression  := unaryExpression
                      ( multiplyOperator
                        unaryExpression
                       )*
unaryExpression    := ( unaryOperator
                        postfixExpression
                      )*
                    | postfixExpression
postfixExpression  := primaryExpression
                      ( ("." | "->") propertyCall )*
propertyCallParam  := "(" declarator?
                          actualParameterList?
                        ")"
```

```
primaryExpression  := literalCollection
                    | literal
                    | propertyCall
                    | "(" expression ")"
                    | ifExpression
literal            := string
                    | number
literalCollection  :=  collectionKind "{"
                       ( collectionItem
                         ("," collectionItem )*
                       )? "}"
collectionItem     := expression (".." expression)?
propertyCall       := lowName
                      timeExpression?
                      qualifiers?
                      propertyCallParam?
qualifiers         := "[" actualParamList "]"
declarator         := lowName ( "," lowName )*
                      ( ":" simpletype )?
                      ( ";" lowName ":" type "=" expression )?
                      "|"
timeExpression     := "@" "pre"
actualParamList    := expression ( "," expression )*
logicalOperator    := "and" | "or" | "xor" | "implies"
relationalOperator := "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator        := "+" | "-"
multiplyOperator   := "*" | "/"
unaryOperator      := "-" | "not"
uppName            := ["A"-"Z"] ( ["a"-"z"] | ["0"-"9"]
                                | ["A"-"Z"] | "_" ) *
lowName            := ["a"-"z"] ( ["a"-"z"] | ["0"-"9"]
                                 | ["A"-"Z"] | "_" ) *
string             := "'" (["a"-"z"] ( ["a"-"z"] | ["0"-"9"]
                                 | ["A"-"Z"] | "_" ) *) "'"
simpletype         := "integer" | "boolean"
                    | "real"    | "string"
collectionKind     := "set"      | "bag"
                    | "sequence" | "collection"
number             := (["0"-"9"])+ ( "." (["0"-"9"])+ )?
                      ( ("e" | "E"} ( "+" | "-")? (["0"-"9"])+ )?
```

2.2 Context-Sensitive Syntax - Restrictions

2.2.1 Class Block

Here context-sensitive restrictions for the textual form of UML class-diagrams are defined.
CSR101 The name of a class must be unique.
CSR102 The name of an association must be unique.
CSR103 The class name must be declared before it is used in an association.
CSR104 The name of an attribute in a class declaration must be unique.
CSR105 An operation in a class declaration must be unique w.r.t. function overloading.
CSR106 If formal parameters of an operation exists then those must be unique in that operation.
CSR107 Role names in association declaration must be unique and different from attributes and operation names in the document.
CSR108 The role name must be declared in an association declaration before it is used in an OCL constraint.
CSR109 If an association is recursive, role names must be declared with its association name.
CSR110 In an association declaration, the association name must be different from associated classes.

2.2.2 OCL Block

Here context-sensitive restrictions for the OCL constraints are defined based on the OCL [Obj03] (Chapter 6) specification.
CSR201 If a context is on a class then the name of the context must be declared as a class name in the Class Block.
CSR202 If a context is on an operation of a class then the name of the class must be declared as a class name in the class block and the name of the operation must be declared as an operation in that class declaration in the class block.
CSR203 In the operation context, if the return type is declared then it should be same as in the operation declaration of the context class in the class block.
CSR204 In the operation context, the types of the formal parameters of the operation must be compatible with the types of the formal parameters of the same operation of context class in the class block.
CSR205 If a context is on a class then an invariant is the only stereotype allowed.
CSR206 If a context is on an operation of a class then stereo types can be any of invariant, pre and post conditions.
CSR207 Let expression allows to define an attribute or an operation that can be used in a constraint. It is only known within that specific constraint, if it is included in an invariant or pre or post condition. These names of the attributes / operations should not conflict with the names of respective attributes / rolenames and operations of that classifier (context class).
CSR208 All variables in a let expression connected with a classifier must be unique.
CSR209 All attributes used in ocl-expressions must be declared in the context class declaration in class block or in a let expression of the same context as a let variable.

CSR210 All operations used in ocl-expressions must be declared in the context class declaration in class block or in a let expression of the same context as a let variable.
CSR211 All OCL expressions must satisfy type conformance, like Integer is subtype of Real.
CSR212 If a constraint is Definition (DEF) constraint, then all variables defined are known in the same context as where any property of the classifier can be used.
CSR213 In a let expression, if the type of the let variable is declared explicitly then it should be the same type as the right hand side expression, else the type of the let variable is the type of the right hand side expression.
CSR214 If a let expression is in DEF constraint then the let variables act as attributes or operations of that classifier.
CSR215 In an if-expression, first expression must be Boolean type, the types of second and third expressions must be same.
CSR216 All OCL expressions that express invariants should be of type Boolean.
CSR217 Expressions for a logical operator must be Boolean type.
CSR218 Expressions for the relational equality operators (i.e $=$ and $<>$) must be same type. And expressions for other relational operators must not be of type String or Boolean.
CSR219 Expressions for an additional operator must be Integer or Real type.
CSR220 Expressions for a multiplicational operator must be Integer or Real type.
CSR221 An expression for a unary minus operator must be Integer or Real type.
CSR222 An expression for a unary "not" operator must be Boolean type.
CSR223 A postfix time expression (i.e "@pre") is only allowed to be used within post-conditions.
CSR224 The keyword "result" is only allowed to be used within postcondition.
CSR225 The arrow operator "$- >$" is only allowed to be used on collection types.

3 How to specify a formal (programming-) language in DSL ?

This section explains briefly about how to specify a formal (programming-) language in Dresden Specification Language (DSL) as an input for the generator system [Bor04], which is developed at TU-Dresden, Germany, to generate a language-based editor. According to this generator system, the person who writes such a specification in DSL is called specifier, the person who uses the generated editor is called user. This section helps such a specifier in specifying in DSL. The context-free syntax and the context-sensitive syntax of the DSL can be found in [Bor04] (Chapter 5)

3.1 Structure of a specification

Let L be the language to be specified. Here the parts of a specification of L are listed by their names and purposes to help the reader to get a general idea of the structure of the specification:

- Abstract syntax - specifies the context-free syntax of L in DSL
- Attribution - specifies the context-sensitive syntax of L by means of the macro attribute grammar formalism [Bor04] [FV98]
- Logic Description of Node Properties - specifies the context-sensitive syntax of L by means of monadic second-order logic formalism (can be combined with attribution)
- Unparsing - specifies external textual representation of the edited object by means of a transformation of its internal representation as (attributed) abstract syntax tree into a text consisting of terminal symbols of the context-free grammar of L augmented by information about the compliance of the context-sensitive syntax of L
- Structural Transformations - specify the possible transformations of the edited object in the structural editing mode of the language-based editor for L
- Parsing - specifies the concrete input syntax of L, i.e the transformation of inputed text during the textual editing mode of the language-based editor for L into a part of the abstract syntax tree internally representing the edited object
- Auxiliary functions - have the purpose of aiding the specification of the last five items

The following subsections describe each of them in brief. Detailed information is available in [Bor04] (Chapter 6).

3.2 Abstract Syntax

Every change of the edited object done by a structural transformation or a textual input changes its abstract syntax tree.

How to specify: To define the underlying structure of the language L context-free syntax (or grammar) is used. Taking such a grammar, by abstracting from the terminal symbols, and giving the productions unique names, one gets a definition of the abstract syntax of L. The edited object in the language-based editor is internally represented by an abstract syntax tree which is built up from this abstract syntax.

Data Block: For the definition of the abstract syntax of L, DSL has the construct of a data block. A data block consists of its name and a list of data definitions encapsulated by the keywords "begindata" and "enddata".

Data Definition: Each data definition defines an algebraic data type. The name of the defined algebraic data type is given in the left hand side of the data definition and in the right hand side a non-empty list of constructors defining the type. Every constructor is followed by a list of its successor types, which are either algebraic data types defined in the same data block or the basic *Haskell* types String, Char, Bool, Int, Double, and Float.

Abstract Syntax Tree: An abstract syntax tree is built up by gluing together constructors of the algebraic data types and elements of the non-algebraic data types in a way that respects their types. An element of a non-algebraic data type can only be used as a leaf.

How Many Data Blocks: In the specification of an language-based editor for L several data blocks are allowed, but only one defines the abstract syntax of L.

Root Type: The data block defining the abstract syntax of L must contain the definition of a special algebraic type, viz the root type. This type has only one constructor and is not allowed in the successor position of any other constructor defined in the data block. By this root type, one can fix that every abstract syntax tree has the same root node.

Initial Abstract Syntax Tree: In a data definition, the first constructor is called default constructor. The initial abstract syntax tree is built by starting with unique constructor of root type and then every successor type by its default constructor. So the specifier must make the order of the constructors such that neither infinite cycles nor non-algebraic data types occur in this process.

3.3 Attribution

This section is to specify context-sensitive syntax of L with a macro attribute grammar.

Mag Block: The DSL construct mag block can be used to specify such a grammar by defining synthesized and inherited functions over the algebraic data types defined in the data block specifying the abstract syntax of L. A mag block is encapsulated in keywords "beginmag" and "endmag".

Connecting the Abstract Syntax: In the input definition of a mag block the connection to the data block can be fixed, i.e the input data block of this mag block. In the root definition of a mag block the root symbol is fixed, viz the unique constructor of root type.

Context-Sensitive Syntax: By means of the synthesized and inherited functions over the abstract syntax trees, context-sensitive syntax of L can be checked. To do so, these functions can produce error messages depending on the violations in the edited object.

Synthesized and Inherited Functions: Every function defined in a mag block must be declared before in this mag block i.e it must be declared as synthesized or inherited with its type signature. The type signature consists of node argument, which is obligatory and node of an abstract syntax tree, optional context parameters, and the result of the function. These attribute functions can be polymorphic, i.e node argument can be more than one type from abstract syntax of L, by using the where-from construct. The function definition is called rule, which can be either with rule head, i.e tree pattern t@(Con t_1 .. t_k) with the keyword **in**. and one or more rule definitions for the specified constructor Con, or a rule on a specific algebraic data type. For example, an inherited attribute function **env**, and a synthesized attribute function **syn** on a constructor **CONS** are shown below :

```
t@(CONS t1 t2) in
 env t1          = env t
 env t2          = env t
 typ t           = typ t1
```

A rule without a rule head can have only one definition

```
typ CONSNULL     = EMPTY
```

These functions must be defined deterministic and non-circular.

3.4 Logic Description of Node Properties

The context-sensitive syntax of L can also be specified by means of a set of Monadic Second-Order (MSO) logic -formulas. It is also possbile to combine mso logic formalism with macro attribute grammar formalism.

Mso Block: The DSL construct mso block can be used to define such formulas over the abstract syntax trees defined in the data block for the abstract syntax of L. The connection to data block can be fixed through input definition of mso block.

Context-Sensitive Syntax: The formulas define so-called node properties with respect to the considered abstract syntax tree with exactly one free node variable. By means of these node properties the context-sensitive syntax of L can be specified. These node properties in the abstract syntax tree can be used to produce error messages depending on the violation of context-sensitive syntax of L in the edited object. Every formula specifying a node property must have a declaration and a definition. A formula declaration consists of the formula name and its type signature. The type signature consists of type of node argument and the result of the formula. The node argument can be one or more types, i.e formula can be polymorphic using where-from construct similar to attribute functions. A formula name in the formula declaration starts with a keyword "?". The result type must be boolean. A formula definition consists of a left-hand side which specifies the formula name and the only free node variable and a right-hand side defining the node property, which is syntactic sugared mso* -formula. In the right-hand side it is possible to use declared formulas.

```
?formula-name :: node-argument -> Bool -- declaration
?formula-name node-variable = mso*-syntactic formula -- defintion
```

3.5 Unparsing

This section defines, how the edited object should be textually represented in the editor. Since the edited object is internally represented as an abstract syntax tree. So the unparsing is a translation function from abstract syntax tree to plain text consisting of terminal symbols of the context-free syntax of L and error-messages concerning violations of context-sensitive syntax of L. The translation function is recursive over the abstract syntax tree and returns **String**. This function is like an attribute function so it is encapsulated in keywords "beginmag" and "endmag", but it must be specified with keyword "unparse" in its obligatory declaration with its input definition to data block and root constructor of data block. The specification must define at least one unparse function and it is possible to have more than one unparse block.

3.6 Structural Transformations

Structural transformations are defined based on the abstract syntax of the language L. The definition of the transform function enables structural editing in the generated language-based editor for L. Just like the unparse function, the transform function can be seen as a special synthesized function respecting some additional restrictions. At most one transform function is allowed to be defined. It must be specified by the keyword "transform" in its obligatory declaration. The mag block defining this transform function must have the same input definition and same root definition as the mag blocks for unparse functions. The transform function specified with declaration and definitions. The declaration starts with keyword "transform", and name of the function then type signature, which has a node argument type, a context parameter and the result type, which should be same as the node argument type.

```
beginmag Blockname_Transform_Mag
 input <input-datablock>
 root  <root-symbol>

 transform name :: node-argument -> contextparameter -> result

 <function definitions>
```

And definitions are set of rules on algebraic data types defined in input definition of the data block. To allow more than one rule definition to be applicable, i.e to have different transformations applicable for a given subtree, the transform function uses the context parameter of type TransSelection, to select one out of a set of different transformations. The type TransSelection has to be defined before by the user as an algebraic data type which contains only nullary constructors. These constructors are then used as the names for the structural transformations invoked by the user of the language-based editor.

3.7 Parsing

The transformations of textual input is specified in this section. To make textual input possible in language-based editor, specification of parsing must contain three blocks, namely a lex block, a parse block and an input block. These three blocks must own the same name.

Lex Block: defines a scanner by a list of tokens specified by means of regular expressions, encapsulated in keywords "beginlex" and "endlex", where the first keyword is followed by the name of the block. The order of token definitions is important for the purpose of scanning.

Parse Block: defines a parser by means of a context-free grammar, using the defined tokens in lex block, which constructs the input syntax tree. This block begins with the keyword "beginparse" followed by its name and a list of parse definitions and ends with the keyword "endparse". A parser for a context-free grammar is specified by means of the parse definitions. A parse definition represents a set of productions of this context-free grammar together with so-called semantic actions used for building the input syntax tree.

Input Block: determines which computational structures of L should be enabled to be textually editable and how, i.e connecting the concrete input syntax of L with abstract syntax of L. The input block begins with keyword "begininput" followed by its name and a list of input determinations, where each of them allows the textual input of a certain type of the concrete input syntax. An input determination consists of an algebraic data type of abstract syntax which is to be edited, then the keyword "as" followed by a parse definition name having the type of algebraic data type. It is possible to restrict the type to one of its constructors using the keyword "@" to precede the algebraic data type

3.8 Auxiliary Functions

Auxiliary functions are used to aid the specification. These are defined as normal Haskell functions. It is possible to define these functions recursively over algebraic data types defined in data blocks. Note that such types are instances of the Haskell type classes Show, Eq, and Read.

4 Specification of the Editor for UML/OCL documents

This section describes the specification of textual form of UML class-diagrams and OCL constraints. This specification is input to the generator system [Bor04] to generate a language-based editor for UML/OCL documents. One can use this editor to write OCL constraints for some set of classes. Since the generator system understands the specification only in DSL format, the specification contains the specification structure of DSL (which is described in the section 3.1)

The following subsections describe the specification in detail. Sometimes as an example the UML class-diagram in the Figure 1 is used to discuss on OCL constraints.

4.1 Abstract Syntax

To define the abstract syntax of UML/OCL documents the context-free grammar of UML/OCL documents is used. The definition of the abstract syntax of UML/OCL documents is obtained by abstracting the grammar from the terminal symbols, and giving the productions unique names. In fact, the object edited in the editor for UML/OCL documents is internally represtend by an abstract syntax tree which is built up by respecting this abstract syntax. The abstract syntax of UML/OCL documents is defined in a single data block. The name of the data block is **Model_Data**.

Each data definition of the data block defines a production of the context-free grammar of UML/OCL documents. For example, consider the first production of context-free grammar of UML/OCL documents from section 2.1.

```
Program  -> "model" uppName classBlock "constraints" oclBlock "endmodel"
```

By omitting the terminal symbols "model", "constraints" and "endmodel", giving the production name as "MODEL", the data definition of above production becomes :

```
data Program           =   MODEL UppName ClassBlock OclBlock
```

Each data definition defines an algebraic data type, which lies in the left hand side of the data definition, and a non-empty list of constructors defining the type are in the right hand side. In the above example, **Program** is the algebraic data type and **MODEL** is the only constructor of algebraic data type **Program**. This constructor has three successor types namely **UppName**, **ClassBlock**, and **OclBlock**, which are again algebraic data types. Similarly, all the productions of the context-free grammar of UML/OCL documents are defined with unique production names and algebraic data types in the data block. In the abstract syntax of UML/OCL documents **Program** is the root type and **MODEL** is the root symbol. The abstract syntax of UML/OCL documents is obtained by starting with root type and gluing together defining the successor types of root symbol. These data definitions are defined such a way that no circular definition occurred in the initial abstract syntax tree.

Initial Abstract Syntax Tree: The initial abstract syntax tree of UML/OCL documents is shown in Figure 3.

The abstract syntax specification has similar style for similar productions. For example, data definitions for all list types has similar specification. i.e ClassList is the algebraic

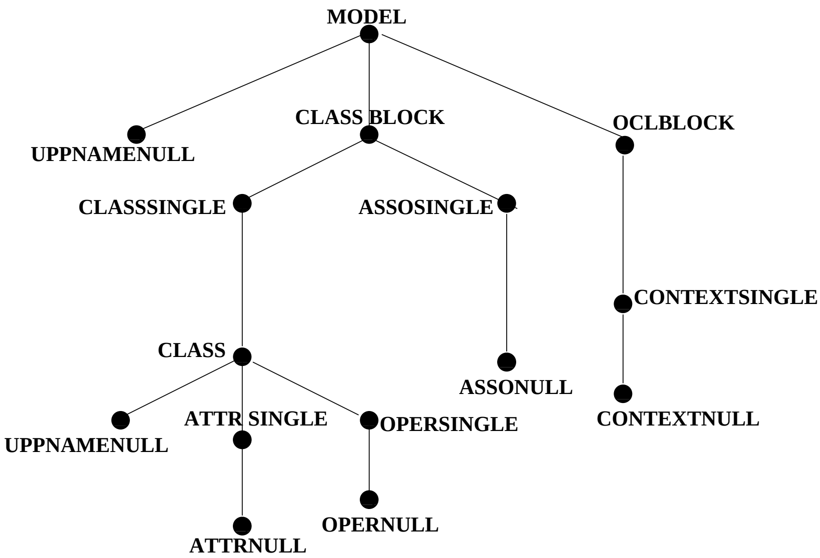

Figure 3: The initial abstract syntax tree

data type of list of classes. ClassList can be single class or it can be pair of a class and a list of classes which is specified as:

```
data ClassList = CLASSSINGLE Class
               | CLASSPAIR  Class ClassList
```

Similarly list of attributes, list of operations, list of parameters, etc. are specified.

Some algebraic data types are defined with constructor HIDE, which is used to hide the details of abstract syntax tree of that algebraic data type from unparsing. In the specification, the algebraic data types Class and Asso are defined with HIDE constructor in the Data Block. The data definition of Class with HIDE constructor is specified as:

```
data Class = CLASS UppName AttrList OperList
           | HIDECLASS UppName AttrList OperList
```

The production simpleType of the context-free grammar is specified with basic types and collection types. These collection types are used in the specification of context-sensitive syntax. For example, in the specification of the synthesized attribute function **has_type**, which is used to check the type of the navigation to association classes (which in general collection type) is collection type. The simpleType is defined as an algebraic data type in the abstract syntax as shown below :

```
data SimpleType = SIMPLETYPENULL
                | INTTYPE  | REALTYPE | BOOLTYPE | STRTYPE
                | COLLTYPE | SETTYPE  | BAGTYPE  | SEQUENCETYPE
```

In the abstract syntax, most of the algebraic data types has default constructors without any successor types. These default constructors are used as place-holders in the editor. The specification consists of two types of place-holders. The first one is for unspecified part of the program, which is to be specified in future. The second one is for unspecified part of the program, which may or may not be specified in future. The place-holder, which is to be specified, is always postfixed with **NULL**. The one which may not be specified is always prefixed with **EMPTY**. In the above specification of SimpleType the default constructor is **SIMPLETYPENULL**. It is used as a place-holder to specify the type.

```
data ReturnType = EMPTYRETURNTYPE
                | RETURNTYPE Type
```

In the above specification, the algebraic data type ReturnType, which can be empty, is defined. The default constructor **EMPTYRETURNTYPE** is used as a place-holder to specify the type.

4.2 Attribution

This section consists of the specification of some of synthesized and inherited functions to check violations of context-sensitive restrictions of UML/OCL documents defined in the section 2.2.

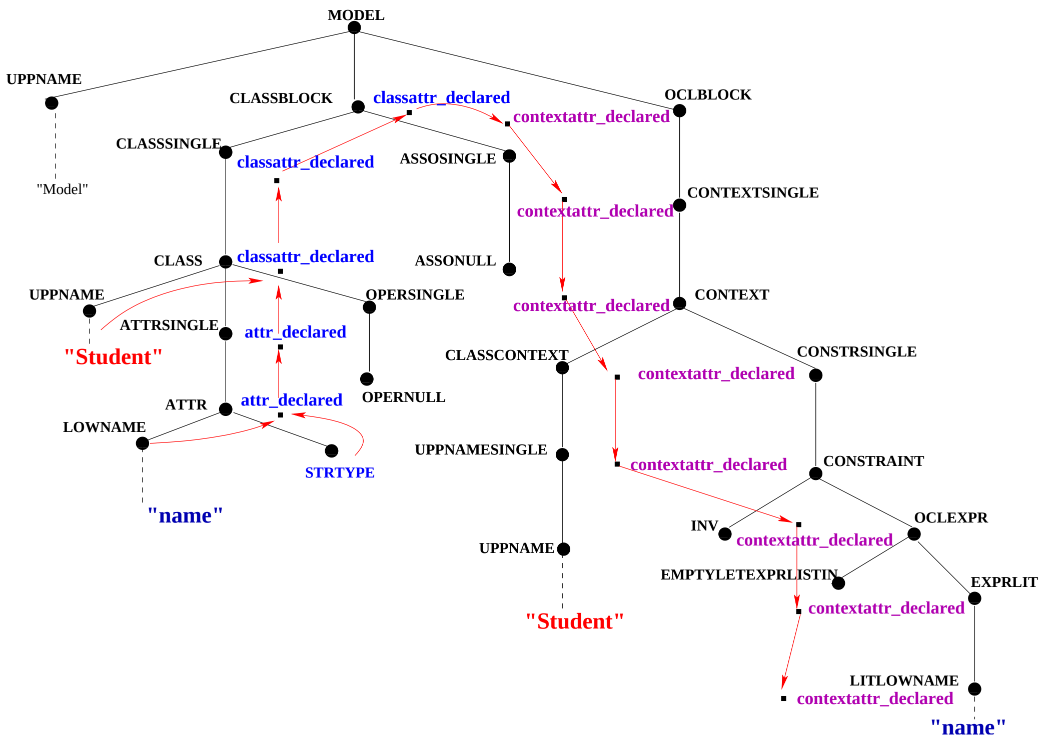

Figure 4: an attributed abstract syntax tree

In Figure 4, it is shown a part of the context-sensitive syntax of the editor, i.e an attribute used in a constraint must be declared in the context class **(CSR209)**. In all figures of this section, the synthesized attributes are shown in blue colour, the inherited attributes are shown in magenta colour and information transport is in red colour. The inherited attribute function **contextattr_declared** is used to transport the information of an attribute from the root **MODEL** to the OclBlock. The synthesized attribute function **classattr_declared** is used to get the information of an attribute to the root **MODEL** from the ClassBlock using another synthesized attribute **attr_declared** inside a class. The inherited function **contextattr_declared** takes three arguments, where the first is the node argument, the second is the context name, which is evaluated with another inherited attribute function **context_name** as shown in Figure 5, and the third is the name of the attribute used in the constraint. The synthesized attribute function **classattr_declared** takes these three arguments and compares the class name with the second argument, if it matches then it checks for an attribute in the third argument inside the attribute declaration of the matched class. If it finds an attribute with same name as in the third argument then returns the type of the attribute, otherwise returns empty type.

In Figure 5, the synthesized attribute function **context** is to get the name of the context from the context declaration. The inherited attribute function **context_name** gets the name of the context from the synthesized attribute **context** at the root **CONTEXT** and transports this information to the constraints.

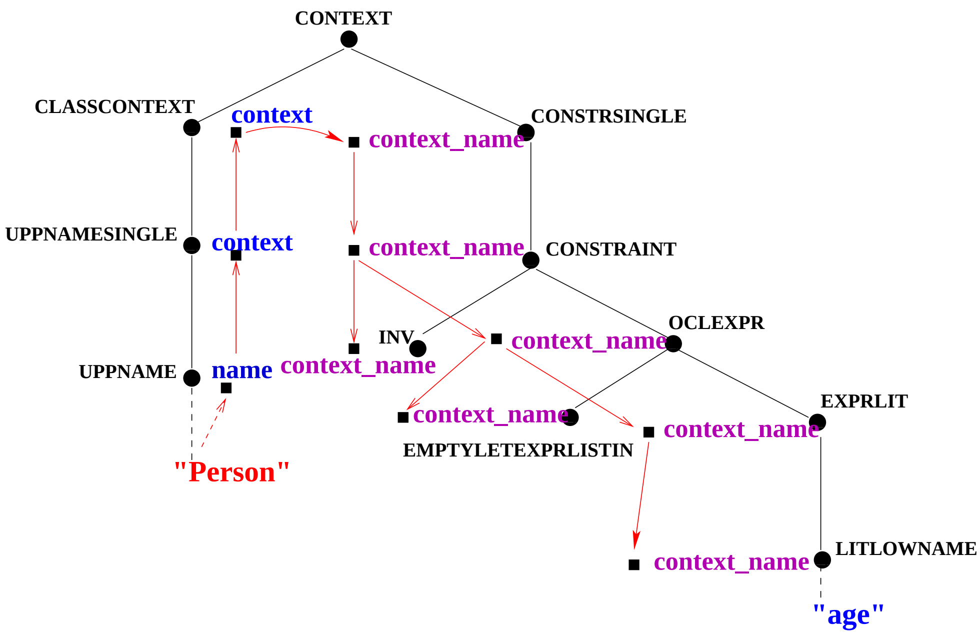

Figure 5: an attributed abstract syntax tree

Similarly, the context-sensitive syntax of **CSR210** is specified.

In Figure 6, a part of the context-sensitive syntax specification is shown in attributed abstract syntax tree, i.e **CSR224**, it checks whether the context is on operation with post-condition. Because the keyword **result** is allowed only in post-condition of an operation.

4.3 Logic Description of Node Properties

This section consists of mso* logic formulas. One can also specify context-sensitive syntax only with mso* logic formulas, but it is too complex to define for all nodes and also editor becomes slow in checking for violations of context-sensitive restrictions. In

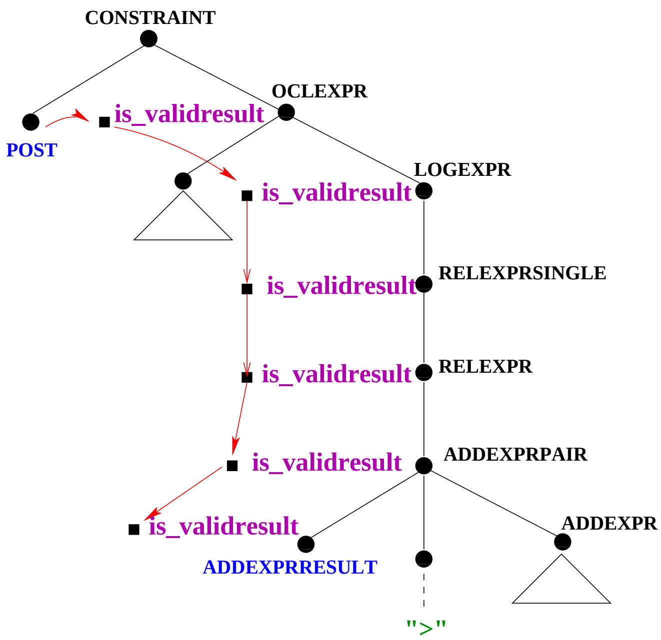

Figure 6: an attributed abstract syntax tree

the specification of UML/OCL documents, the context-sensitive syntax is defined by paradigms of both attribution and logic description of node properties. In the input definition of mso block, the connection to the data block **Model_Data** is fixed.

Some mso* logic formulas of the specification are ?role_declared, ?in_roledecl and ?is_validstereotype. Each formula with its purpose is described below.

?in_roledecl : is used to check whether a **LowName** is declared as a role. This formula is used as a subformula in different logic formula. This formula shown below can be read as, if a free node-variable x has **ROLELOWNAME** as parent then it is true, where Parent is a predefined formula in DSL.

```
formula ?in_roledecl     :: LowName -> Bool

?in_roledecl x     = Parent x ROLELOWNAME Node y (True)
```

?role_declared : is used to check whether role is declared before it is used in the document (CSR108). It takes the node argument as **LowName**. This formula is defined using another mso* formula called **?in_roledecl** to test the node variable is in role declaration, an auxiliary function **equal_name** to test two strings are equal in names, and a synthesized attribute function **name** to get the name of the node variable.

```
equal_name :: String -> String -> Bool
equal_name = (==)

syn name :: LowName -> String
name LOWNAMENULL  = ""
name (LOWNAME t1) = t1
```

The formula shown below can be read as, if a free node-variable x is labeled as **LOWNAME**, then there exists a node y of type **LowName**, which is in role declaration and names of x and y are equal.

```
formula ?role_declared       :: LowName -> Bool

?role_declared x
        = Implies
             (Label LOWNAME x)
             (ExNode y :: LowName
                (And
                   (?in_roledecl y)
                   (Mag
                      (equal_name
                         (name x)
                         (name y)))))
```

?is_validstereotype : is used to check whether a **Stereotype** is valid in OCL Block (CSR205-CSR206). Especially pre- and post- conditions are allowed only for a context on an operation. The formula shown below can be read as, if the free node-variable x is labeled with either **PRE** or **POST**, then if there exist a node z of type **Context**, then if there exist an another node y of type **ContextDecl**, then y is labeled with **OPERCONTEXT** and the node x and the node y are under the node z in the abstract syntax tree.

```
formula ?is_validstereotype :: StereoType -> Bool

?is_validstereotype x
        = Implies
             (Or
                (Label PRE x)
                (Label POST x))
             (ExNode z :: Context
                (ExNode y :: ContextDecl
                   (And
                      (Label OPERCONTEXT y)
                      (And
                         (Under x z)
                         (Under y z)))))
```

These logic formulas define a part of the context-sensitive syntax. It is also possible to define context-sensitive syntax without mso block (i.e logic formalism), but logic formalism makes specification simpler. And it is good to use logic formalism paradigms which makes the specification simpler.

4.4 Unparsing

This section is about the specification of textual representation of abstract syntax of UML/OCL documents defined in the data block **Model_Data**. The definition part of the unparse function **up** starts with the only constructor **MODEL** of root type **Program** as shown below, then it is defined recursively over the successor types of root symbol declared in the where-from construct.

```
unparse up :: a -> String where a from {Program, ....}

up (MODEL t1 t2 t3) = "model "       ++ (up t1) ++ "\n" ++
                                        (up t2) ++ "\n" ++
                      "constraints" ++ (up t3) ++ "\n" ++
                      "endmodel"
```

In the above rule of the unparse function, the terminal symbols "model", "constraints" and "endmodel" are recovered that are lost in the translation of the context-free syntax into the abstract syntax. Similarly all other terminal symbols can be also recovered. In the unparse function definitions the operator ++ is used to concatenate strings, and control signs `"\n", "\t", and "\b"` are used to define indentations and linebreaks in the textual representation of the edited object.

The unparse function is defined also to display the error messages on violations of context-sensitive restrictions using some attribute functions and auxiliary functions in the editor while an object is being edited in the editor. For example, the specification of unparse function definition for algebraic data type **Attr** (i.e attribute declaration of a class) consists also the specification of checking for violations of unique attribute and if attribute is declared then its type must be declared (**CSR104**) as shown below.

```
unparse up :: Attr -> String

up ATTRNULL          = " <attribute> "
up (ATTR t1 t2)      = (up t1) ++
                        (check " {multiple attribute name} "
                           ((t1 == LOWNAMENULL) ||
                            (not (fst (declared1 t t1))))) ++
                        (up t2)
```

Since **Attr** type has two constructors, unparse function is defined on two constructors. The first constructor **ATTRNULL** is a place holder (i.e unspecified and to be specified) in the editor, so right hand side of unparse function rule consists "<attribute>", to inform the user that it is place holder. The second constructor **ATTR** is the abstract syntax of attribute declaration. The unparse function is defined for this constructor, recursively on its successor types, i.e **t1 and t2**, with displaying error messages on violations of context-sensitive restrictions using two auxiliary functions **check, and is_empty**, and one inherited attribute function **declared1**.

In the unparse block, the specification of displaying errormessages on violations of context-sensitive restrictions consists logic formulas also. For example checking whether the context is on operation to use post- or pre- conditions is shown below.

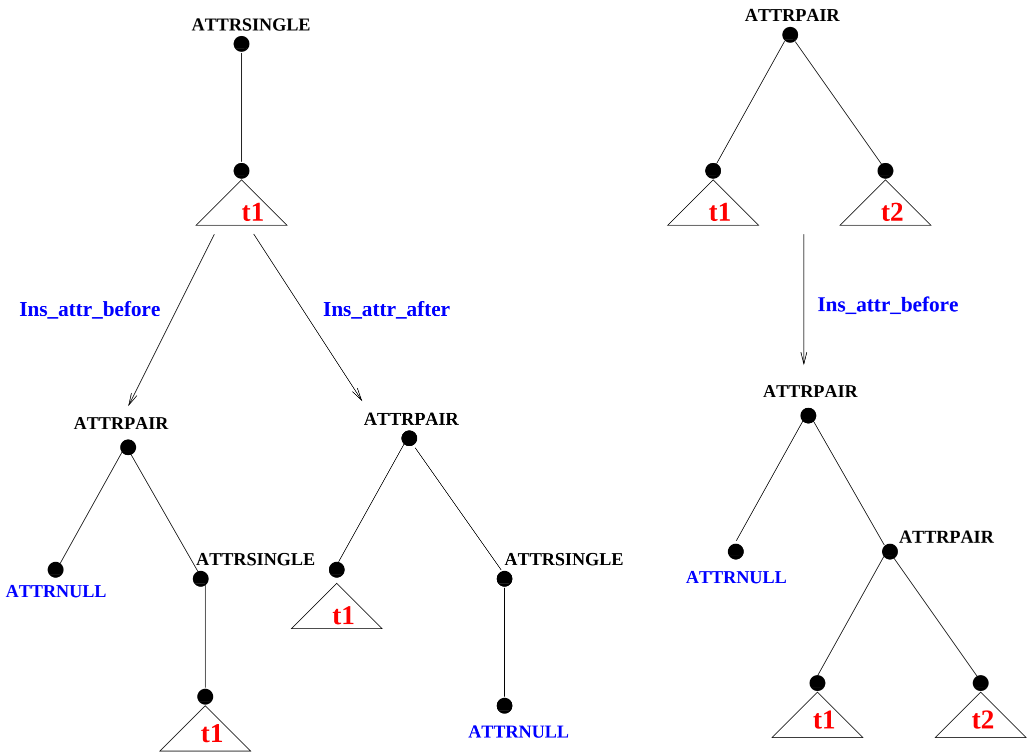

Figure 7: some abstract syntax tree transformations of type AttrList

```
up (CONSTRAINT t1 t2 ) = (up t1) ++
                         (check " {context is not an operation } "
                            (?is_validstereotype t1)) ++ ":\t\n" ++
                         (up t2) ++
                         (check " {boolean expression expected } "
                            ((t1 == STEREOTYPENULL) ||
                            (is_boolean_compatible (has_type t3))))
```

4.5 Structural Transformations

This section is about specification of structural transformations on the abstract syntax of UML/OCL documents, which is defined in the data block **Model_Data**. These structural transformations are defined on the abstract syntax using the transform function **trans**.

In the editor for UML/OCL documents, an object can be edited in two modes. One is the structural editing mode and the other is the textual input mode using keyboard. The transform function **trans** is specified to enable the structural editing mode. This transform function is defined for all algebraic data types, especially for all place-holders defined in the input data block.

For example, in the specification of UML/OCL documents, **SIMPLETYPENULL** is the place-holder for the type **SimpleType**, which stands for unspecified type. Then transform function is defined on this place-holder using the context parameters **Ins_Integer** to transform into **Integer** type and **Cut** to transform **Integer** type into empty type, as shown below.

```
transform trans :: Type -> TransSelection -> Type

trans SIMPLETYPENULL Ins_Integer = INTTYPE
trans INTTYPE        Cut         = SIMPLETYPENULL

data TransSelection = Ins_Integer | Cut | ..
```

Similarly, the transform function is specified for all algebraic data types which needs to be edited in the editor for UML/OCL documents. In the specification, some patterns have similar definitions. For instance, Ins_before, Ins_after, Cut_first and Cut_rest are constructors defined in TransSelection. They are used as context parameters for all node argument of list types. For example, possible transformations on list of attributes in a class declaration are shown below. It can be seen in abstract syntax trees as shown in Figure 7, where inserted objects are in blue colour.

```
transfrom trans :: AttrList -> TransSelection -> AttrList

t@(ATTRSINGLE t1) in
 trans t  Ins_attr_before = (ATTRPAIR ATTRNULL t)
 trans t  Ins_attr_after  = (ATTRPAIR t1
                             (ATTRSINGLE ATTRNULL))))

t@(ATTRPAIR t1 t2) in
 trans t  Ins_attr_before = (ATTRPAIR ATTRNULL t)
 trans t  Ins_attr_after  = (ATTRPAIR t1 (ATTRPAIR ATTRNULL t2))
 trans t  Cut_first       = t2
 trans t  Cut_rest        = (ATTRSINGLE t1)

data TransSelection = ..
                    | Ins_attr_before | Ins_attr_after
                    | Cut_first       | Cut_rest | ..
```

4.6 Parsing

This section contains the specification of transformation of textual input via keyboard into a part of the abstract syntax tree of edited object in the editor for UML/OCL documents. To make textual input possible in the editor, three blocks are defined in the specification for three different purposes with the same name **Model_input**. These three blocks are described in next three sub-sections.

4.6.1 Lexical Analysis

This block defines the scanner by the list of tokens specified by means of regular expressions. There are two type of tokens:

- non-value-capturing tokens just matching a text - are used as keywords, as shown below, it just matches to string "Integer".

```
token    IntgerT = "Integer"
```

- value-capturing tokens which matches a text and stores a value of the specified type, as shown below, it matches to regular expression in the right hand side via the *Haskell* read-facilities. i.e Int can be any integer 0,1,....99....etc..

```
token   IntValueT Int  = '0' | [1-9] [0-9]*
```

The order of the token definitions are important for the process of scanning. The token **Whitespace** is specified to discard the text during the scanning process. Some token definitions of the specification of editor for UML/OCL documents are listed below.

```
token Whitespace      = [\ \t\n]
token ClassT          = "class"
token AssoT           = "association"
token ContextT        = "context"
token LetT            = "let"
token IfT             = "if"
token ColonT          = ":"
token AndT String     = "and"
token OrT String      = "or"
token UppNameT String = [A-Z] [0-9a-zA-Z_]*
token LowNameT String = [a-z] [0-9a-zA-Z_]*
..
```

4.6.2 Concrete Input Syntax

This section describes the specification for parsing the textual input with the help of the tokens defined in the lex block. This specification is in DSL construct called parse block. The parser is defined for the context-free grammar (defined in section 2.1) by the list of parse definitions. A parse definition represents a set of productions of this context-free grammar. For example, a parse definition of type attribute is defined below.

```
parse P_Attr :: Attr
| P_LowName
  ColonT
  P_SimpleType     = ATTR $1 $3
```

A parse definition begins with a keyword **parse** followed by a non-terminal symbol i.e **P_Attr** here, which is to be specified context-free production. **Attr** is an algebraic data type of the attribute for the non-terminal symbol **ATTR**, which can store a part of input syntax tree. In the above example, in the right hand side of the definition consists the semantic definition built up by constructor **ATTR** of the attributed type **Attr** together with the keywords $1 and $3, which are the number of symbols between | and =. Similarly the parser is defined for some parts of the context-free syntax of textual form of UML class-diagrams and for the complete context-free syntax of OCL constraints.

The parser for the parts of context-free syntax of textual form of UML class-diagrams is defined such a way that the textual input is possible for name of a class, and complete structures of associations, role-names, attributes and operations. The parser is defined for all OCL constraints. For example, the parser definition for a context is shown below.

```
parse P_Context       :: Context
 | ContextT
   P_ContextDecl
   P_ConstraintList  =     CONTEXT $2 $3
 | ContextT
   P_ConstraintList  =     CONTEXT CONTEXTDECLNULL  $2
```

Where **P_Context** is the parser definition of algebraic data type **Context** defined in the abstract syntax, **ContextT** is a token for keyword "context", and **P_ContextDecl, P_ConstraintList** are parser definitions for context declaration and list of constraints respectively.

4.6.3 Input Determination

In this section, input determination of the parser is specified. i.e which constructs of the abstract syntax should be textually editable and in which way this should be done. This is done in an input block. It contains a list of input determinations. The input determination starts with the keyword **input**. It is followed by an algebraic data type of the abstract syntax, followed by the keyword **as**, and followed by the name of parse definition. For example:

```
input Attr as P_Attr
input EMPTYTYPE @ Type as P_Type
```

In the above example, **Attr** is the algebraic data type of the data block for attribute and **P_Attr** is the parse definition for the type attribute in the concrete syntax. It is possible to restrict the type to one of its constructors using the keyword **@** to precede the type as shown in the above example.

It is possible to use an auxiliary function in the input determination to transform the input syntax tree to an element of the specified type of the abstract syntax. Some input determinations of input block of the specification are shown below. The input determination for name of the model has used an auxiliary function **complete_name**.

```
begininput Model_Input
 input MODEL.1 @ UppName as P_UppName via complete_name
 input UppName as P_UppName
 input LowName as P_Lowname
 input Context as P_Context
 input ContextDecl as P_ContextDecl
 input ConstraintList as P_ConstraintList
 ..
endinput
```

4.7 Auxiliary Functions

This section consists some auxiliary functions which are used in other parts of specification. These functions are normal *Haskell* functions. In the specification of editor for UML/OCL documents, the **Char** type is imported to use in the definition of an auxiliary function. Some functions are defined on the algebraic data types of the data block. Some of the auxiliary functions of the specification are listed below.

```
check :: String -> Bool -> String
```

The function **check** is used to generate error messages depending on the violations of context-sensitive restrictions of UML/OCL documents. It takes error message as the first argument and a predicate as the second argument. The predicate can be built up for instance by attribute or auxiliary functions, node properties (i.e mso* logic formulas), or equality tests involving parts of the abstract syntax tree. If the predicate is evaluated to be false, then the result of this function is the error- message, otherwise it is the empty string.

```
complete_name :: UppName -> UppName
```

The function **complete_name** takes a name, which starts with upper case letter and returns the same name decorated with *, it is used just to differentiate from other names. This function is used in first input determination of input block.

```
is_empty :: SimpleType -> Bool
```

The function **is_empty** is used to check whether given type is empty. It takes algebraic data type **SimpleType** as the argument, which is often built up by synthesized and inherited functions. If the argument is equal to **SIMPLETYPENULL** then the **is_empty** function returns true, otherwise returns false.

```
is_boolean_compatible :: SimpleType -> Bool
```

The function **is_boolean_compatible** is used to check whether the given type is empty or boolean. It takes the algebraic data type **SimpleType** defined in the data block and returns true if it is boolean or empty.

```
getType :: LowName -> LowName -> (FormalParams,SimpleType)
           -> (FormalParams,SimpleType)
```

The function **getType** is used to get the type of an operation which is declared either in the operations declaration of a class or in a let expression of a constraint. It takes three arguments, where the first is the name of the operation, the second is name of an operation used in a constraint, and the third is the formal parameters and return type of the operation. If the first and second arguments are equal (i.e same names) then the function **getType** returns the formal parameters and return type of the operation, otherwise returns empty formal parameters and empty simple type.

```
compareTypes :: FormalParams -> TypeList -> Bool
```

The function **compareTypes** is used to compare the types of an operation used in the expression of a constraint with the one declared in a class or in a let expression. It takes two arguments, where the first is first argument of result of the function **getType** and the second is the types of actual parameters of the operation, which is used in constraint. If the first and the second has same number of types and same types then returns true, otherwise returns false.

```
lower :: String ->String
```

The function **lower** takes a string, returns the same string after changing the first letter to lower case using a predefined function **toLower** in **Haskell**. The predefined function **toLower** takes **Char** type as input and returns **Char** type. So specification needs to import **Char** type. This function is used in comparing an association name with a role name for the synthesized attribute **roleclass_name** in the association declaration of the class block.

5 Conclusions

This project has built a specification of the textual form of UML class-diagrams, and OCL constraints in DSL. This specification is used to generate a language-based editor for UML/OCL documents using the generator system. For this purpose, the specification of OMG-UML v1.5 is used. The editor can be used to program some set of textual form of UML class-diagrams and some set of OCL constraints on these class-diagrams. The editor makes sure that the program edited is correct with respect to the context-free syntax and checks for violations of the context-sensitive restrictions. In the textual form of UML class-diagrams only basic types are allowed as types for the attributes, the operations, and the formal-parameters of the operations. For OCL constraints, the editor recognizes most of the predefined operations on collection types. In the editor the enumerated types, predefined properties on all objects (i.e. oclIsKindOf, oclInState, etc..), and the package construct of OCL specification are not defined.

A User Manual

Together with this report a disk is placed. It contains the source code of the specification file i.e Editor.dsl. It also contains three example files i.e Course.eo, Family.eo and Employee.eo. The program Course.eo is in Figure 1 and the program Family.eo is in Figure 2. The following subsections describe about how to start the editor and how to use the editor.

A.1 How to start the editor ?

To start the editor, the user has to type **Start**. in the console (eg: $:/home/Start). Now the user can see the editor which consists of a template as shown in the Figure 8, and the user can edit that template. The editor window consists of three parts. The first part is the place with "File" pop-up menu in the center, which consists of New, Open, Save and Exit options. A user can view these options by clicking on the "File" pop-up menu. A program can be saved in .eo format (eg: temp.eo), which stands for edited-object. The second part consists of the place to edit the program, most of the space in the window is occupied by this part with two scrollbars. The third part consists of several possible buttons to edit the program structurally and the textbox which shows the type of the current edited object of the program in the editor.

A.2 How to use the editor ?

text highlighting: In the editor, initially the whole program is highlighted in red. If the user selects an object then current edited object is highlighted in red, and the remaining in blue.

The user can edit the program in two modes, one is by textual input via keyboard, and the other is by structural editing using possible buttons in the third part of the editor.

textual input mode: It is possible to type the whole program by textual input using keyboard, but user is allowed to type in a single line. The user must be careful in editing via textual input, for example, OCL keywords are invalid as names of class,

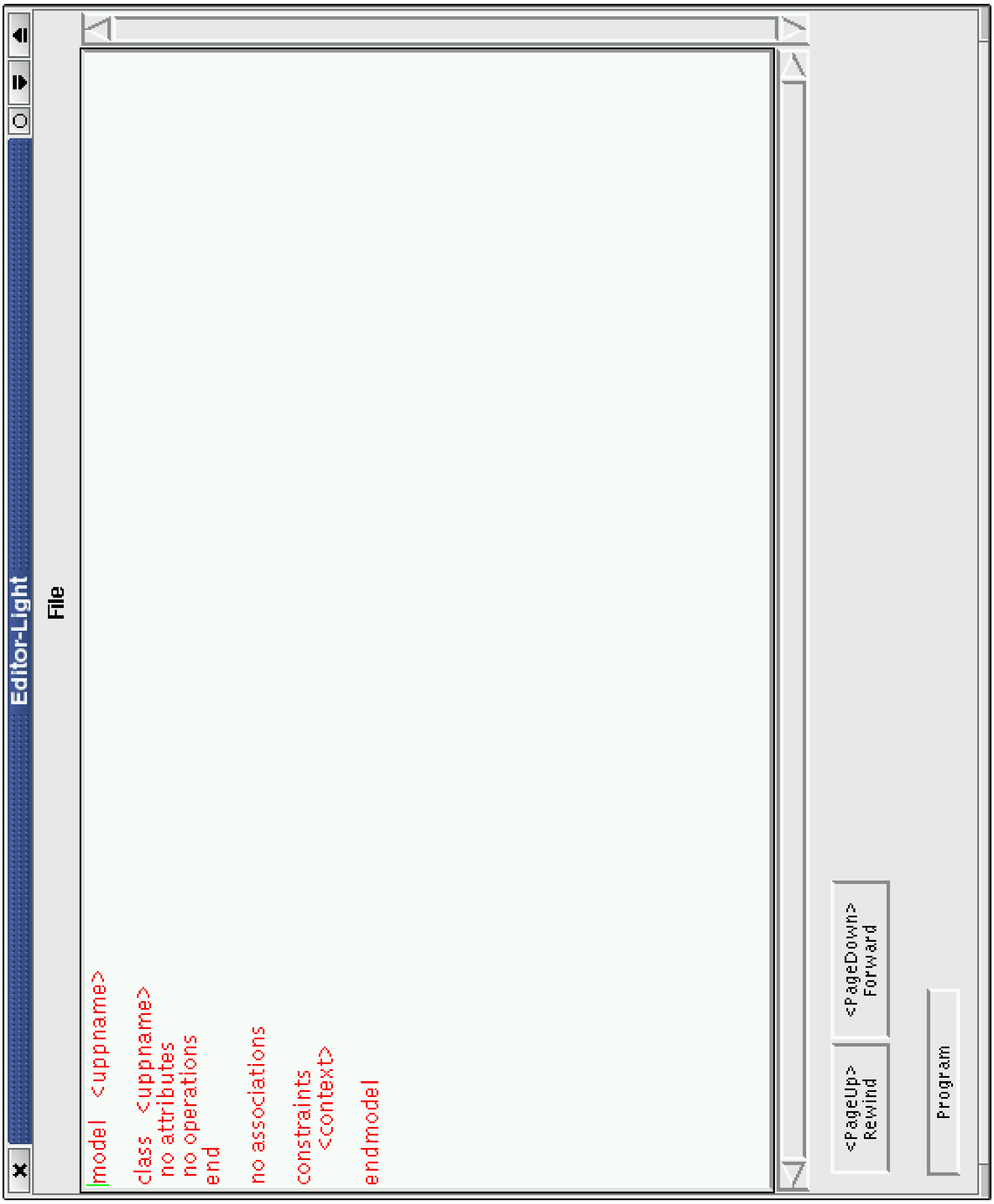

Figure 8: Editor for UML/OCL documents

association, attribute, operation, rolename, etc., i.e. the keyword "class" is invalid for attribute name. If there is an error in parsing, editor will not accept the whole input. So the user must be careful in the textual mode of input. One can insert text by pressing **insert** key at the keyboard, which erases whole text of the editing object. And a user can select an object of the program in the editor by clicking with mouse on it. If some part of the program is not accepted by the editor via textual input mode, then one can try in structural mode. In the editor, some parts of the program has to be entered only in textual inputing mode via keyboard only. The editor displays empty names to help the user in editing the names, for example <uppname> is to indicate the name which starts with an uppercase letter. The unspecified objects, which can be entered through textual input mode are listed below.

```
<uppname>      : name starting with upper case letter
<lowname>      : name starting with lower case letter
<literal>      : which can be either string of characters enclosed
                 in single quotation marks (i.e 'xxx') or number
<logoper>      : logical operator (i.e and, or , implies, (or) xor)
<reloper>      : relation operator (i.e >, <, = , >=, <=, (or) <>)
<addoper>      : additive operator plus or minus (i.e +, (or) -)
<muloper>      : multiplicative operator (i.e *, (or) /)
<unaryoper>    : unary minus operator (i.e -) or "not" operator
```

structural mode: If a user is not familiar with syntax of the document then one can edit the whole program by structural editing mode. And also one can learn the syntax of the UML/OCL documents easily by structural editing. Initially the editor consists default template. Whenever the user wants to edit some part of the program structurally, the user can make use of set of possible buttons of the selected object. These buttons are displayed in the third part of the editor. In fact, the program in the editor for UML/OCL documents is internally in the tree format, a user can move forward or backward in the tree by clicking on buttons with mouse or by pressing <Page Down> or <Page UP> buttons respectively at the keyboard. These two buttons are always available to move forward and backward in the editor. Buttons of different functionality are shown below with few examples.

```
Ins_X          : inserts an X structure, where X can be an attribute,
                 an operation, an association etc..
Ins_X_before   : inserts an X structure before an existing one
Ins_X_after    : inserts an X structure after an existing one
Cut            : erases the selected object and displays
                 default constructor of selected type
Cut_first      : cuts the first object of selected structure
Cut_rest       : cuts the rest of the objects from
                 the selected object in a structure
Hide_details   : hides details of the selected object
Display_details: displays hidden details of the selected object
Ins_navig      : inserts navigation structure to access properties
                 e.g.: self.a->size()
```

```
Ins_logoper    : inserts two expressions with a logical operator
                 e.g.: a and b, (or) a implies b, etc..
Ins_reloper    : inserts two expressions with a relational operator
                 e.g.: a > b, a <> b, etc..
Ins_add        : inserts two expressions with a + operator
                 e.g.: a + b
Ins_sub        : inserts two expressions with a - operator
                 e.g.: a - b
Ins_mul        : inserts two expressions with a * operator
                 e.g.: a * b
Ins_div        : inserts two expressions with a / operator
                 e.g.: a / b
Ins_coll       : inserts collection of literals structure
                 e.g.: Set { 1, 4, 3}, Sequence{ 1 .. 3}
```

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison–Wesley, 1986.

[Bor04] E. Bormann. Automatic Generation of Language-Based Editors using DSL, 2004. Manuscript.

[FV98] Zoltán Fülöp and Heiko Vogler. *Syntax-directed semantics - Formal models based on tree transducers*. Monographs in Theoretical Computer Science. Springer-Verlag, Berlin/Heidelberg, first edition, 1998.

[Obj03] Object Management Group, Inc. Unified Modeling Language (UML) version 1.5. Available on: http://www.omg.org/technology/documents /formal/uml.htm, March 2003.

[RT89] T. W. Reps and T. Teitelbaum. *The synthesizer generator — a system for constructing language–based editors*. Springer–Verlag, 1989.

[FHP98] Joseph Fasel, Paul Hudak, John Peterson.
A Gentle Introduction to Haskell, Version 1998.