Implementation of a Syntax-Checker for an MSO-Logic over Trees

Endrawaty

Project Report
International Master Programme
Computational Logic
Oct 24, 2003 - Apr 23, 2004

Professur Grundlagen der Programmierung
Prof. Dr.-Ing. habil. Heiko Vogler
Fakultät Informatik
Technische Universität Dresden

Supervisor:
Dipl.-Inf. Enrico Bormann

Contents

1 Introduction

In the spirit of easing the phase of program development, one might use a specialized editor that is designed for editing a certain formal (programming) language. This editor should be able to tell whether the object being edited contains error or not. Moreover, it should be able to inform in which part of the edited object the errors actually occur. It works interactively by giving error information back and forth to the user of the editor as the object being modified.

To create such a language-based editor for a formal (programming) language, we might use a generator system [2, 3] which takes the specification of the language as input, and then generates the language-based editor as output. The generator works just like a compiler, scanning and parsing the specification of the language. It also performs context-sensitive restrictions checking against that specification. If it finds some errors, then it returns the error information. If it finds no error, then it translates the specification into another structure which will run the language-based editor.

Let L be the name of the language to be specified. A specification of L mainly consists of its abstract syntax and its context-sensitive syntax. We call the person who writes such a specification as a specifier. In this project context, we use DSL [2] (Dresden Specification Language) as the specification language. DSL provides three formalisms to specify a formal language. Those formalisms are functional programming, macro attribute grammar, and monadic second-order (mso)-logic over abstract syntax trees. The specifier should combine those three formalisms to get the most effective and efficient specification of L. In a brief explanation, a specification consists of the following structures [2]:

- **abstract syntax** - specifies the structure of L by utilizing a sorted ranked alphabet related to a context-free grammar defining the context-free syntax of L,

- **attribution** - specifies the context-sensitive syntax of L by using the macro attribute grammar formalism,

- **logic description of node properties** - specifies the context-sensitive of L by using the mso-logic formalism,

- **unparsing** - specifies the 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, along with information about the conformance of the context-sensitive restrictions 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 input syntax of L by means of a 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, and

- **auxiliary functions** - are used for completing the specification of the last five structures.

To define the underlying structure of L, we use the concept of context-free grammar. By omitting the terminal symbols of the context-free grammar and giving unique names to the productions, we get the definition of abstract syntax of L. Here is an example of how to abstract from a production of a context-free grammar of a language into its abstract syntax:

- Consider the following production of a context-free grammar:

$$Program \rightarrow \text{"program"}\ Identifier\ \text{";var"}\ DeclLs\ \text{"begin"}\ StmtLs\ \text{"end."}$$

- Omit the terminal symbols of the production and give the production the name PROG, then we get an element of the sorted ranked alphabet defining the abstract syntax of the language:

$$PROG^{(Identifier\ DeclLs\ StmtLs, Program)}$$

The object edited by the language-based editor for L is internally represented by an abstract syntax tree which is built up respecting the abstract syntax of L. For specifying the abstract syntax of L, DSL has the construct of a *data block*. A data block consists of a name and a list of data definitions. Each data definition defines an *algebraic data type* which specifies some elements of the sorted ranked alphabet defining the abstract syntax.

The context-sensitive syntax of L can be specified by macro attribute grammar and mso-logic formalisms. Here we use an extension of mso-logic, denoted by mso*-logic, which uses an attribute grammar based semantics for the logic. DSL combines the two concepts with the concept of functional programming style.

In macro attribute grammar, the nodes of the abstract syntax tree are enriched with attributes. The DSL construct *mag block* can be used to specify such a grammar by defining functions over the algebraic data types defined in the data block specifying the abstract syntax of L. By means of those functions, information transfer over the the abstract syntax trees can be defined, which makes it possible to check the context-sensitive syntax of L.

The concept of mso*-logic over trees provides the abstract syntax trees of L with a set of mso*-formulas. The DSL construct *mso block* can be used to define such formulas over the abstract syntax trees. Each of these formulas has only one free node variable, through which the formula defines the so-called *node properties*. By these node properties, the context-sensitive syntax of L is specified in a logical and global way.

With macro attribute grammar, information transfer can be done in a flexible way, upward and downward along the the nodes of the abstract syntax tree. While with functional programming, information transfer is naturally done upward, for the reason that only the nodes on the lower part can be used as argument for a function which is called by the function defined on the nodes of the upper part. With mso*-logic over trees, the specifier will be able to globally specifiy a certain node of the abstract syntax tree in the intuitive way as a human mind would work. That is why the combination of these three facilities in DSL can be used to specify the context-sensitive syntax of L.

The generator system designed and implemented in [1] already has the scanner and parser for the specification of L. It also has the syntax checker for the context-sensitive restrictions specified by macro attribute grammar formalism. But it still needs the syntax checker for mso*-logic formalism. This checker will receive the parsed logic part of the specification of L as input. It will also receive the parsed data block and the parsed function enviroment defined in the attribution, unparsing and structural transformations structures of the specification. As a result, it will give information about the context-sensitive syntax violation in the logic part of the specification as output. The goal of this project is the development and implementation of this syntax checker using the functional programming language Haskell. This syntax-checker should be integrable into the existing generator system in [1].

This project report is organized as follows. In Section 2, we define context-free grammars, sorted sets, sorted ranked alphabets, sorted terms, sorted trees, abstract syntax, and macro attribute grammars. In Section 3, we explain about the mso*-logic over trees and its syntax restriction. In Section 4, we explain the implementation of the syntax checker for the mso*-logic over trees. In Section 5, we draw conclusions about this project.

2 Preliminaries

The underlying theoritical concepts in this project consist mainly of macro attribute grammar and mso*-logic formalisms. They are needed specify a formal language using DSL. To define macro attribute grammar, we need first to introduce the concept of context-free grammars, sorted ranked alphabets, and abstract syntax trees. Furthermore, for defining sorted ranked alphabets, we need to define first what a sorted set is. While for defining abstract syntax trees, it is necessary to previously define the concept of directed-labeled graphs, sorted terms and sorted trees. We will discuss about mso*-logic and its syntax restriction in the next section.

2.1 Context-Free Grammars

Definition 1. (Context-Free Grammar).
A *context-free grammar* is a tuple $G_0 = (N, \Lambda, Z, P)$, where :

- N is a finite set of *non-terminal symbols*,
- Λ is a finite set of *terminal symbols* (we state: $N \cap \Lambda = \emptyset$),
- $Z \in N$ is the *start symbol*,
- P is a finite set of *productions* of the form

$$\eta_0 \to \lambda_0 \eta_1 \lambda_1 \ldots \eta_l \lambda_l$$

  with $l \geq 0$, $\eta_0, \ldots, \eta_l \in N$, and $\lambda_0, \ldots, \lambda_l \in \Lambda^*$.

We call G_0 *start-separated* if Z occurs exactly in one production, and moreover it occurs there at the left-hand side. □

2.2 Sorted Sets, Sorted Ranked Alphabets, Directed Labeled Graph, Sorted Terms, and Sorted Trees

Definition 2. (S-sorted Set).
Let S be a finite set of sorts. An *S-sorted set* is a pair $(A, sort)$ where A is a set and $sort : A \to S$ is a mapping which assigns to every $a \in A$ a sort $s \in S$.

An element $a \in A$ with $sort(a) = s$ is denoted by a^s. For every sort $s \in S$, A^s denotes the subset of A that contains all elements a^s: $A^s = \{a \mid a \in A \wedge sort(a) = s\}$. When $sort$ is known from the context, we abbreviate $(A, sort)$ by A. □

Definition 3. (S-sorted Ranked Alphabet).
Let S be a finite set of sorts. An *S-sorted ranked alphabet* is a pair $(\Sigma, rank)$ where Σ is a finite $(S^* \times S)$-sorted set and $rank : \Sigma \to \mathbb{N}$ is a mapping which assigns to every $\sigma \in \Sigma^{(s_1 \ldots s_l, s_0)}$ with $l \geq 0$ and $s_0, \ldots, s_l \in S$ the rank l.

For every rank $l \geq 0$, Σ_l denotes the subset of Σ that contains all elements σ with rank l: $\Sigma_l = \{\sigma \mid \sigma \in \Sigma \wedge rank(\sigma) = l\}$. If the mapping $rank$ is known from the context, we abbreviate $(\Sigma, rank)$ by Σ.

We define the function $sort_{res} : \Sigma \to S$ such that for every $\sigma \in \Sigma^{(s_1 \ldots s_l, s_0)}$ with $l \geq 0$ and $s_0, \ldots, s_l \in S$, $sort_{res}(\sigma) = s_0$. We define the maximal rank of a ranked alphabet as $max\_rk\_\Sigma = max(\{rank(\sigma) \mid \sigma \in \Sigma\})$.

□

Definition 4. (Directed Labeled Graph).
Let Σ and Γ be alphabets (of node labels and edge labels, respectively). A *directed labeled graph* over (Σ, Γ) is a tuple (V, E, lab), where:

- V is the finite set of nodes,
- $E \subseteq V \times \Gamma \times V$ is the finite set labeled edges, and
- $lab : V \to \Sigma$ is the node-labelling function.

If $(v, i, w) \in E$, then we say that v has (v, i, w) as outgoing edge and w has (v, i, w) as incoming edge. For a given graph g, the set of its nodes, set of its edges, and its node-labelling function are denoted by V_g, E_g, and lab_g, respectively. □

Definition 5. (S-sorted Term).
Let S be a finite set of sorts, Σ be an S-sorted ranked alphabet and Y be an S-sorted set. The *set of all S-sorted terms over* Σ *indexed by* Y, denoted by $T_\Sigma(Y)$, is the smallest S-sorted set $\bigcup_{s \in S} T^s$ such that:

1. $Y^s \subseteq T^s$, and
2. for every $\sigma \in \Sigma^{(s_1 \ldots s_l, s_0)}$ with $l \geq 0$ and $s_0, \ldots, s_l \in S$ and for every $te_1 \in T^{s_1}, \ldots, te_l \in T^{s_l}$, it holds that $\sigma(te_1, \ldots, te_l) \in T^{s_0}$.

We abbreviate $T_\Sigma(\emptyset)$ by T_Σ. □

Every S-sorted term over Σ can be represented in a graphical form by a set of S-sorted trees over Σ. We define S-sorted tree in the following definition.

Definition 6. (S-sorted Tree).
Let S be a finite set of sorts and Σ be an S-sorted ranked alphabet. An *S-sorted tree* t over Σ is directed labeled graph over Σ and $\{1, \ldots, max\_rk\_\Sigma\}$ such that:

1. there is exactly one node without incoming edges, which we denote by $root(t)$,

2. for every node $n \in V_t$ with incoming edges, there is exactly one sequence $n_0, \ldots, n_h$ with $h \geq 1$, $n_0 = root(t)$, and $n_h = n$ such that for every $j \in [1..h]$, $(n_{j-1}, i_j, n_j) \in E_t$ with $i_1, \ldots, i_h > 0$, and

3. V_t is an S-sorted set such that for every node $n \in V_t$, the following holds:

   - if $lab(n) = \sigma \in \Sigma^{(s_1 \ldots s_l, s_0)}$ with $l \geq 0$, $s_0, \ldots, s_l \in S$ then $n \in V_t^{s_0}$ and
   - for every $i \in [1..l]$ there is exactly one node $n_i \in V_t^{s_i}$ such that $(n, i, n_i) \in E_t$.

The set of all S-sorted trees over an S-sorted ranked alphabet Σ is denoted by $TR(\Sigma)$. For every $t \in TR(\Sigma)$, we define $sort(t) = sort(root(t))$. □

2.3 Abstract Syntax

Abstract syntax of a formal (programming) language L can derived from the productions of a context-free grammar of the language L. In the following, we define the relatedness between context-free grammars and sorted ranked alphabets, then we define the concept of abstract syntax trees.

Definition 7. (Relatedness between a Context-Free Grammar and a Sorted Ranked Alphabet).
Let $G_0 = (N, \Lambda, Z, P)$ be a context-free grammar and π be a bijection $\pi : P \to \Sigma$ where Σ is a ranked alphabet with cardinality $|P|$ of *production names*. We call Σ related to G_0 *by* π if it is an N-sorted ranked alphabet such that the following holds:

- For every production $p = (\eta_0 \to \lambda_0 \eta_1 \lambda_1 \ldots \eta_l \lambda_l \in P$, with $l \geq 0$, $\eta_0 \ldots \eta_l \in N$, $\lambda_0 \ldots \lambda_l \in \Lambda^*$, and $\pi(p) = \sigma \in \Sigma$, we have $sort(\sigma) = (\eta_1 \ldots \eta_l, \eta_0)$.

□

Definition 8. (Abstract Syntax Tree).
Let $G_0 = (N, \Lambda, Z, P)$ be a (start-separated) context-free grammar and Σ be the N-sorted ranked alphabet which us related to G_0 by a bijection $\pi : P \to \Sigma$. The set of abstract syntax trees of G_0 regarding π, denoted by $AST(G_0, \pi)$, is the set of all N-sorted trees $TR(\Sigma)^Z$. If G_0 is start-separated then the consequences are:

- there is exactly one symbol $\sigma \in \Sigma$ with $sort_{res}(\sigma) = Z$ and
- for every abstract syntax tree $t \in TR(\Sigma)^Z$, $lab_t(root(t)) = \sigma$, and for every node $n \in V_t$ not being the root of t:
$$lab_t(n) \neq \sigma$$

□

2.4 Macro Attribute Grammars

Macro attribute grammar formalism is used to specify the context-sensitive restriction of a formal language L. In macro attribute grammar, the nodes of an abstract syntax tree are enriched with attributes. Basically, macro attribute grammar is a context-free grammar extended by :

- attaching "attributes" to the nonterminal symbols of the grammar and

- supplying "attribute rules" to define attribute values.

In macro attribute grammar, the attributes of the nonterminal symbols can be assigned values or functions.

Definition 9. (Macro Attribute Grammar).
A *macro attribute grammar* is a tuple $M = ((G_0, \pi), D, B, R)$ consisting of:

- a *start-separated context-free grammar* G_0 together with a bijection π,
- a *semantic domain* D, which defines function symbols in macro attribute grammar,
- an *attribute description* B, which defines attribute symbols in macro attribute grammar, and
- a set R of *semantic rules*.

The descriptions of each element of the tuple of macro attribute grammar are given below:

- G_0 is a start-separated context-free grammar (N, Λ, Z, P), π is a bijection $\pi : P \to \Sigma$ with $|P| = |\Sigma|$ such that Σ is the N-sorted ranked alphabet related to G_0 by π.
- D is a tuple $(K, \Omega, \mathcal{F}, int_{\mathcal{F}})$ consisting of
  - a finite set K of sorts,
    with $N \subseteq K$
  - a K-sorted set Ω (*set of semantic values*),
    where for every $\eta \in N, \Omega^{\eta} = TR(\Sigma)^{\eta}$
  - a finite K-sorted ranked alphabet $\mathcal{F}$ of *function symbols*,
    with $\Sigma \subseteq \mathcal{F}$, and
  - an *interpretation function* $int_{\mathcal{F}}$ that assigns to every function symbol $f \in \mathcal{F}$ a function over subsets of Ω:

    For every $k \geq 0$ and $\kappa_0, \ldots, \kappa_k \in K$ the following holds: for every $f \in \mathcal{F}^{(\kappa_1 \ldots \kappa_l, \kappa_0)}$, $int_{\mathcal{F}}(f) : \Omega^{\kappa_1} \times \ldots \times \Omega^{\kappa_k} \to \Omega^{\kappa_0}$ is a k-ary computable function,
    where for every $\sigma \in \Sigma^{(\eta_1 \ldots \eta_l, \eta 0)} \subseteq \mathcal{F}$ with $l \geq 0$ and $\eta_0 \ldots \eta_l \in N \subseteq K$, $int_{\mathcal{F}}(\sigma) : \Omega^{\eta_1} \times \ldots \times \Omega^{\eta_l} \to \Omega^{\eta_0}$ is the l-ary computable function which takes l trees of the specified sorts $\eta_1, \ldots, \eta_l$ and creates a new tree of the sort η_0 with a new node as root labeled by σ using the given arguments as the l subtrees.

- B is a tuple (*S-Att, I-Att*), having the following properties:
  - *S-Att* is a finite $(\mathcal{P}(N) \times K^* \times K)$-sorted set whose elements are called *synthesized macro attributes.*
  - *I-Att* is a finite $(\mathcal{P}(N - \{Z\}) \times K^* \times K)$-sorted set whose elements are called *inherited macro attributes.*
  - *S-Att* and *I-Att* are disjoint.

– For every $\bar{\eta} \in \mathcal{P}(N)$, $r \geq 0$, $\kappa_0, \ldots, \kappa_r \in K$, and $a \in (S\text{-}Att \cup I\text{-}Att)^{(\bar{\eta}, \kappa_1 \ldots \kappa_r, \kappa_0)}$, we define the functions:

$$sort_{nodes} : (S\text{-}Att \cup I\text{-}Att) \to \mathcal{P}(N) \text{ and}$$
$$sort_{fun} : (S\text{-}Att \cup I\text{-}Att) \to (K^* \times K)$$

such that

$$sort_{nodes}(a) = \bar{\eta} \text{ and}$$
$$sort_{fun}(a) = (\kappa_1 \ldots \kappa_r, \kappa_0).$$

– For every production $p = \eta_0 \to \lambda_0 \eta_1 \lambda_1 \ldots \eta_l \lambda_l \in P$ with $\pi(p) = \sigma \in \Sigma^{(\eta_1 \ldots \eta_l, \eta_0)}$, the K-sorted ranked alphabet of *attribute occurences of the production p named by* σ is the set:

$$A(\sigma) = \{\langle a, j\rangle^{sort_{fun}(a)} \mid a \in (S\text{-}Att \cup I\text{-}Att) \wedge 0 \leq j \leq l \wedge \eta_j \in sort_{nodes}(a)\}$$

- R is a pair (Y, RP) consisting of:
  - a finite K-sorted set Y of *context parameters* and
  - a family $RP = \{R(\sigma) \mid \sigma \in \Sigma\}$ of *semantic rules.*

□

3 MSO*-Logic

In this section, we define an extension of mso-logic, denoted by mso*-logic. It extends mso-logic by integrating macro attribute grammar formalism into the logic. This will relate us to the definition we have given in Definition 9 about macro attribute grammar.

Mso*-logic can be used to specify the context-sensitive syntax of L by means of a set of mso*-formulas. The formulas are defined over the abstract syntax trees in the DSL construct of mso block. Those formulas have exactly one free node variable, that is a variable which denotes a node in the abstract syntax tree. This means that the formulas define node-properties of the abstract syntax tree. With these node properties, the context-sensitive syntax of L is specified in a logical and therefore more global way. In the following subsections, we discuss the mso*-logic syntax and its restriction.

3.1 MSO*-Logic Syntax

First, we define the following symbols:

Let N and K be finite sets of sorts.
Let $\mathcal{F}$ be a K-sorted ranked alphabet of function symbols.

The definitions of N, K, and $\mathcal{F}$ follow exactly what is given in Definition 9.

3.1.1 Macro Attribute Symbols

Let $\mathcal{A}$ be a $(\mathcal{P}(N) \times K^* \times K)$-sorted set of macro attribute symbols. In relation with Definition 9, $\mathcal{A}$ is the union of S-Att and I-Att. We will use $\mathcal{A}$ in the folowing definitions.

Definition 10. (Macro Attribute Variable Symbols).
Let N be a finite set of sorts and $\mathcal{A}$ be a $(\mathcal{P}(N) \times K^* \times K)$-sorted set (of macro attribute symbols). Let $\mathcal{V}_1$ be a $\mathcal{P}(N)$-sorted set (of node variables). For $\mathcal{V}_1$, we define the K-sorted ranked alphabet (*of macro attribute variable symbols*):

$$\mathcal{I}\langle \mathcal{A}, \mathcal{V}_1 \rangle = \{ \langle a, x \rangle^{sort_{fun}(a)} \mid a \in \mathcal{A} \ \wedge \ x \in \mathcal{V}_1 \ \wedge \ sort(x) \subseteq sort_{nodes}(a) \}$$

□

Definition 11. (Macro Attribute Instance Symbols).
Let N be a finite set of sorts and $\mathcal{A}$ be a $(\mathcal{P}(N) \times K^* \times K)$-sorted set (of macro attribute symbols). Let V be an N-sorted set (of nodes). For V, we define the K-sorted ranked alphabet (*of macro attribute instance symbols*):

$$\mathcal{I}\langle \mathcal{A}, V \rangle = \{ \langle a, n \rangle^{sort_{fun}(a)} \mid a \in \mathcal{A} \ \wedge \ n \in V \ \wedge \ sort(n) \in sort_{nodes}(a) \}$$

□

3.1.2 Formulas

Let Σ be an N-sorted ranked alphabet. Let $\mathcal{V}_1$ be a $\mathcal{P}(N)$-sorted set of node variables. Let $\mathcal{V}_2$ be a $\mathcal{P}(N)$-sorted set of node set variables. The set $MSO^*(\Sigma, \mathcal{V}_1, \mathcal{V}_2)_{\mathcal{A},\mathcal{F}}$ is the set consisting of the following formulas:

- $true$,
  (this formula always holds)
- $label_\sigma(x)$,
  where $x \in \mathcal{V}_1$, $\sigma \in \Sigma^{(\eta_1 \ldots \eta_l, \eta_0)}$ with $l \geq 0$ and for every $j \in [0..l], \eta_j \in N$ such that $\eta_0 \in sort(x)$,
  (this formula tests if the node x is labeled by the symbol σ)
- $type_\eta(x)$,
  where $x \in \mathcal{V}_1$ and $\eta \in sort(x)$,
  (this formula tests if the node x has the sort η)
- $x \in X$,
  where $x \in \mathcal{V}_1$, $X \in \mathcal{V}_2$, and $sort(x) = sort(X)$,
  (this formula tests if the node x is an element of the set of nodes X)
- $x == y$,
  where $x, y \in \mathcal{V}_1$ and $sort(x) = sort(y)$,
  (this formula tests if the nodes x and y are equal)
- $edge_i(x, y)$,
  where $x, y \in \mathcal{V}_1$ and $i > 0$,
  (this formula tests if the node y is the i-th successor of the node x)
- $x \ over \ y$,
  where $x, y \in \mathcal{V}_1$,
  (this formula tests if there is a path through the tree which leads from the node x to the node y)

- $\varphi \in T^{Bool}_{\mathcal{I}\langle\mathcal{A},\mathcal{V}_1\rangle\cup\mathcal{F}}$
  (this formula tests if this (well-sorted) term φ consisting of macro attribute instance symbols (with node variable argument) and function symbols can be interpreted as boolean value $True$. **Note**: $\mathcal{B}ool \in K$)

- $\neg\psi$,
  where $\psi \in MSO^*(\Sigma, \mathcal{V}_1, \mathcal{V}_2)_{\mathcal{A},\mathcal{F}}$,
  (this formula tests if the formula ψ does not hold)

- $\psi_1 \wedge \psi_2$,
  where $\psi_1, \psi_2 \in MSO^*(\Sigma, \mathcal{V}_1, \mathcal{V}_2)_{\mathcal{A},\mathcal{F}}$,
  (this formula tests if both formulas ψ_1 and ψ_2 hold)

- $\exists x \ : \ \bar{\eta} \,.\, \psi$, where $x \notin \mathcal{V}_1$ and $\psi \in MSO^*(\Sigma, \mathcal{V}_1 \cup \{x^{\bar{\eta}}\}, \mathcal{V}_2)_{\mathcal{A},\mathcal{F}}$,
  (this formula tests if there is a node x with a sort of the set of sorts $\bar{\eta}$, such that the formula ψ holds)

- $\exists X \ : \ \bar{\eta} \,.\, \psi$, where $X \notin \mathcal{V}_2$ and $\psi \in MSO^*(\Sigma, \mathcal{V}_1, \mathcal{V}_2 \cup \{X^{\bar{\eta}}\})_{\mathcal{A},\mathcal{F}}$
  (this formula tests if there is a set of nodes X, each with a sort of the set of sorts $\bar{\eta}$, such that the formula ψ holds)

- $child_{(\sigma,i)}(x)y \,.\psi$, where $\sigma \in \Sigma^{(\eta_1\cdots\eta_l,\eta_0)}$ with $l \geq i$ and for every $j \in [0..l]$, $\eta_j \in N$, such that $x \in \bigcup_{\bar{\eta}\ni\eta_0} \mathcal{V}_1^{\bar{\eta}}$, $y \notin \mathcal{V}_1$, and $\psi \in MSO^*(\Sigma, \mathcal{V}_1 \cup \{y^{\{\eta_i\}}\}, \mathcal{V}_2)_{\mathcal{A},\mathcal{F}}$,
  (this formula tests if the node x is labeled by σ, and moreover, if in this case the formula ψ holds where y is the i-th successor node of the node x - the sort of the node y is determined by the sort of σ, viz η_i)

- $parent_\sigma(x)y \,.\psi$, where $\sigma \in \Sigma^{(\eta_1\cdots\eta_l,\eta_0)}$ with $l \geq 1$ and for every $j \in [0..l]$, $\eta_j \in N$, such that $x \in \bigcup_{\bar{\eta}\cap\{\eta_1,\ldots,\eta_l\}\neq\emptyset} \mathcal{V}_1^{\bar{\eta}}$, $y \notin \mathcal{V}$, and $\psi \in MSO^*(\Sigma, \mathcal{V}_1 \cup \{y^{\{\eta_0\}}\}, \mathcal{V}_2)_{\mathcal{A},\mathcal{F}}$,
  (this formula tests if there exists a parent node of the node x is labeled by σ, and moreover, if in this case the formula ψ holds where y is the parent node - the sort of the node y is determined by the sort of σ, viz η_0)

3.2 Definition of DSL part for MSO*-Logic

In this section, we will give the definition of DSL part related with mso*-logic. But before we do that, we will give summary of the relation between mso*-logic formula, DSL syntax of mso*-logic and context-sensitive restrictions of mso*-logic. We have discussed mso*-logic formulas in the previous subsection. While the context-sensitive restrictions of mso*-logic will be given in the coming subsection. The summary will uncover the two most important relations as follows:

- Given an mso*-logic formula, then we give the DSL syntax for that formula.

- Given an mso*-logic formula, then we give the context-sensitive restrictions which are applied for that formula.

We classify the formulas into three categories such as atomic formulas, boolean connectives formulas, and quantifiers formulas. These three categories are depicted in the following three tables:

Atomic Formulas:

Formula	*DSL-syntax*	*CSRs*
$true$	`True`	-
$false$	`False`	-
$label_\sigma(x)$	`Label` σ `x`	613, 628
$x \in X$	`IsElem x y`	614, 628
$x == y$	`Equal x y`	615, 628
$type_s(x)$	`TypeOf x s`	616, 628
$edge_i(x, y)$	`Edge x i y`	617, 628
$x\ over\ y$	`Over x y`	628
$x\ under\ y$	`Under x y`	628
$\varphi \in T^{Bool}_{\mathcal{I}\langle\mathcal{A},\mathcal{V}_1\rangle\cup\mathcal{F}}$	`Mag` (φ)	618

Boolean Connectives Formulas:

Formula	*DSL-syntax*	*CSRs*
$\neg\psi$	`Not` (ψ)	-
$\psi_1 \wedge \psi_2$	`And` (ψ_1) (ψ_2)	-
$\psi_1 \vee \psi_2$	`Or` (ψ_1) (ψ_2)	-
$\psi_1 \rightarrow \psi_2$	`Implies` (ψ_1) (ψ_2)	-
$\psi_1 \leftrightarrow \psi_2$	`Iff` (ψ_1) (ψ_2)	-

Quantifiers:

Formula	*DSL-syntax*	*CSRs*
$\exists x : \bar{\eta}.\ \psi$	`ExNode x :: a where a from` $\bar{\eta}$ (ψ)	620, 621, 622
	`ExNode x ::` η (ψ) $(if\ \bar{\eta} = \{\eta\})$	610, 611, 619
$\forall x : \bar{\eta}.\ \psi$	`AllNodes x :: a where a from` $\bar{\eta}$ (ψ)	620, 621, 622
	`AllNodes x ::` η (ψ) $(if\ \bar{\eta} = \{\eta\})$	610, 611, 619
$\exists X : \bar{\eta}.\ \psi$	`ExNodeSet x :: a where a from` $\bar{\eta}$ (ψ)	620, 621, 622
	`ExNodeSet x ::` η (ψ) $(if\ \bar{\eta} = \{\eta\})$	610, 611, 619
$\forall X : \bar{\eta}.\ \psi$	`AllNodeSets x :: a where a from` $\bar{\eta}$ (ψ)	620, 621, 622
	`AllNodeSets x ::` η (ψ) $(if\ \bar{\eta} = \{\eta\})$	610, 611, 619
$child_{(\sigma,i)}(x)\ y.\ \psi$	`Child x` σ `i Node y` (ψ)	610, 611, 623, 628
$parent_\sigma(x)\ y.\ \psi$	`Parent x` σ `Node y` (ψ)	610, 611, 624, 628

Now we come to main part of this subsection, that is the context-free syntax of DSL for mso*-logic. The nonterminal symbols are enclosed between $\langle$ and $\rangle$ and the terminal symbols are enclosed between " and ". The start symbol for mso*-logic is $\langle MsoBlock\rangle$.

The definition is classified into two part, first is the main definition of DSL part for mso*-logic, second is the accompanying definition which is needed by the main definition. The main definition will only be found in mso*-logic specification, while the accompanying definition will also be found in other parts of specification.

3.2.1 The Main Definition

⟨*MsoBlock*⟩	::=	"`beginmso`" ⟨*upperid*⟩ ⟨*InputDef*⟩ ⟨*MsoDef*⟩
		{⟨ *MsoDef*⟩} "`endmso`"
⟨*MsoDef*⟩	::=	⟨*MsoTypeDef*⟩ | ⟨*FormulaDef*⟩
⟨*MsoTypeDef*⟩	::=	"`formula`" ⟨*questid*⟩ ⟨*TypeSig*⟩
⟨*FormulaDef*⟩	::=	⟨*questid*⟩ ⟨*lowerid*⟩ "`=`" ⟨*Formula*⟩
⟨*Formula*⟩	::=	⟨*boolconst*⟩ | ⟨*Base*⟩ | ⟨*questid*⟩ ⟨*lowerid*⟩ |
		"`Mag`" "(" ⟨*RuleRhs*⟩ ") | ⟨*Connection*⟩ |
		⟨*Quantification*⟩
⟨*Base*⟩	::=	"`Label`" ⟨*upperid*⟩ ⟨*lowerid*⟩ |
		"`IsElem`" ⟨*lowerid*⟩ ⟨*lowerid*⟩ |
		"`Equal`" ⟨*lowerid*⟩ ⟨*lowerid*⟩ |
		"`TypeOf`" ⟨*lowerid*⟩ ⟨*upperid*⟩ |
		"`Edge`" ⟨*lowerid*⟩ ⟨*intconst*⟩ ⟨*lowerid*⟩ |
		"`Over`" ⟨*lowerid*⟩ ⟨*lowerid*⟩ |
		"`Under`" ⟨*lowerid*⟩ ⟨*lowerid*⟩ |
⟨*Connection*⟩	::=	"`Not`" ⟨*ClasFor*⟩ |
		"`And`" ⟨*ClasFor*⟩ ⟨*ClasFor*⟩ |
		"`Or`" ⟨*ClasFor*⟩ ⟨*ClasFor*⟩ |
		"`Implies`" ⟨*ClasFor*⟩ ⟨*ClasFor*⟩ |
		"`Iff`" ⟨*ClasFor*⟩ ⟨*ClasFor*⟩
⟨*Quantification*⟩	::=	"`ExNode`" ⟨*lowerid*⟩ ⟨*TypeAss*⟩ ⟨*ClasFor*⟩ |
		"`AllNodes`" ⟨*lowerid*⟩ ⟨*TypeAss*⟩ ⟨*ClasFor*⟩ |
		"`ExNodeSet`" ⟨*lowerid*⟩ ⟨*TypeAss*⟩ ⟨*ClasFor*⟩ |
		"`AllNodeSets`" ⟨*lowerid*⟩ ⟨*TypeAss*⟩ ⟨*ClasFor*⟩ |
		"`Child`" ⟨*lowerid*⟩ ⟨*upperid*⟩ ⟨*intconst*⟩ "`Node`"
		⟨*lowerid*⟩ ⟨*ClasFor*⟩ |
		"`Parent`" ⟨*lowerid*⟩ ⟨*upperid*⟩ "`Node`" ⟨*lowerid*⟩
		⟨*ClasFor*⟩
⟨*TypeAss*⟩	::=	"`::`" (⟨*upperid*⟩ | ⟨*lowerid*⟩ ⟨*TypeRestr*⟩)
⟨*ClasFor*⟩	::=	"(" ⟨*Formula*⟩ ")"

3.2.2 The Accompanying Definition

⟨*Type*⟩ ::= ⟨*upperid*⟩
⟨*InputDef*⟩ ::= "input" ⟨*upperid*⟩
⟨*TypeSig*⟩ ::= "::" (⟨*upperid*⟩ | ⟨*lowerid*⟩) "->" ⟨*Type*⟩ { "->" ⟨*Type*⟩ } ["->" ⟨*lowerid*⟩] [⟨*TypeRestr*⟩]
⟨*TypeRestr*⟩ ::= "where" ⟨*lowerid*⟩ "from" "{" ⟨*upperid*⟩ { "," ⟨*upperid*⟩ } "}"
⟨*RuleRhs*⟩ ::= (⟨*lowerid*⟩ | ⟨*questid*⟩ | ⟨*upperid*⟩) { ⟨*RhsExp*⟩ } | ⟨*RhsExp*⟩ ⟨*InfixOp*⟩ ⟨*RhsExp*⟩ {⟨*InfixOp*⟩ ⟨*RhsExp*⟩ } | ⟨*preinfixop*⟩ ⟨*RhsExp*⟩ { ⟨*InfixOp*⟩ ⟨*RhsExp*⟩ } | ⟨*Const*⟩ | ⟨*Compound*⟩
⟨*InfixOp*⟩ ::= ⟨*infixop*⟩ | ⟨*preinfixop*⟩
⟨*RhsExp*⟩ ::= ⟨*lowerid*⟩ | ⟨*upperid*⟩ | ⟨*Const*⟩ | ⟨*Compound*⟩
⟨*Const*⟩ ::= ⟨*boolconst*⟩ | ⟨*intconst*⟩ | ⟨*charconst*⟩ | ⟨*stringconst*⟩ | "[]"
⟨*Compound*⟩ ::= "(" ⟨*RuleRhs*⟩ { "," ⟨*RuleRhs*⟩ } ")" | "[" ⟨*RuleRhs*⟩ { "," ⟨*RuleRhs*⟩ } "]"
⟨*infixop*⟩ ::= "==" | "/=" | "<" | ">" | "<=" | ">=" | ":" | "++" | "&&" | "||" | "+" | "*" | "‘div‘" | "‘mod‘"
⟨*preinfixop*⟩ ::= "-"
⟨*boolconst*⟩ ::= "True" | "False"
⟨*intconst*⟩ ::= ("0" | ... | "9") { "0" | ... "9" }
⟨*charconst*⟩ ::= "'" (⟨ printable character except "'" and "\" ⟩ | "\" ⟨ printable character ⟩) "'"
⟨*stringconst*⟩ ::= """ { ⟨ printable character except """ and "\" ⟩ | "\" ⟨ printable character ⟩ } """
⟨*upperid*⟩ ::= ("A" | ... | "Z") { "A" | ... | "Z" | "a" | ... | "z" | "0" | ... | "9" | "_" | "'" }
⟨*lowerid*⟩ ::= ("a" | ... | "z") { "A" | ... | "Z" | "a" | ... | "z" | "0" | ... | "9" | "_" | "'" }
⟨*questid*⟩ ::= "?" ⟨*lowerid*⟩

3.3 Restrictions of MSO*-Logic Syntax in DSL

Now, we discuss about the restrictions of mso*-logic syntax in DSL. There are totally 28 syntax restrictions, which can be classified into three categories, viz general restrictions (CSR601 - CSR602), restrictions on formula declarations part (CSR603 - CSR609), and restrictions on mso* formula definitions (CSR610 - CSR628).

3.3.1 General Restrictions

CSR601. The input definition must use a defined data block.

In *DSL*, there can be several data blocks defined. The mso block construct must use one of those data blocks as its input. This choice is done in the input definition ⟨*InputDef*⟩ of mso block.

CSR602. For every formula definition there must be a corresponding declaration (type definition) in the same mso block.

This restriction is quite clear. We also find such restriction in most programming languages which require a variable / function to be declared before it is defined.

3.3.2 Restrictions on Formula Declaration

CSR603. The node argument position of a type signature must be either the name of a type defined in the input data block or a type variable.

The node argument is the first and the only argument which should be given to a formula. It defines at which nodes of an abstract syntax tree the node property can be tested. In a type signature, the type of the node argument can be given directly by mentioning a type defined in the input data block or by using a type variable followed by a where-from type restriction. When it is given directly, that type should be defined in the input data block. Here is an example of a formula declaration which uses `Identifier` as the node argument type:

```
formula ?var_declared :: Identifier → Bool
```

CSR604. In the type signature of an mso-formula only* `Bool` *is allowed as result type.*

This is clear. A logic formula is a boolean function which must always return value of type `Bool`.

CSR605. The type signature of an mso-formula must not define context parameters.*

This means that there should only be two part of a type signature. The first is the node argument type and the second is the result type. No other parameter is allowed.

CSR606. A where-from type restriction must be used if and only if a type variable is used in the node argument position of a type signature.

As we explain in CSR603, the type of node argument can be given directly or by using a type variable. This restriction cares only for the latter case. When a type variable is given, it should be followed by a *where-from* construct. A where-from construct is used to specify polymorphic node properties, i.e. node properties that are defined for more than one type in the node argument. Here is an example of a formula declaration which specifies polymorphic node properties using a where-from construct:

```
formula ?test :: a → Bool where a from { Identifier, Expression }
```

CSR607. The type variable used in a where-from type restriction must be the same as used in the node argument position in the corresponding type signature.

In this restriction, we only care for formulas which use a type variable as their node argument and have a where-from type restriction for that type variable. Here is an example of a formula declaration which violates this restriction:

```
formula ?test :: a → Bool where b from { Identifier, Expression }
```

The cause of violation in the above formula declaration is the use of different type variables "a" and "b". "b" is used in the where-from type restriction, while "a" is used in the node argument position.

CSR608. The types used in a where-from type restriction must be defined in the input data block.

This is clear.

CSR609. In a where-from type restriction no repetition of type names is allowed.

This is also clear.

3.3.3 Restrictions on Formula Definition

CSR610. A variable cannot be a node variable and a node set variable in the same mso-formula (right-hand side of a formula definition).*

Here is an example formula which violates CSR610:

```
?wrong_formula x
  = And (ExNode y :: Identifier (Label IDENTIFIER y))
        (ExNodeSet y :: Identifier (IsElem x y))
```

The above formula violates CSR610 because the variable "`y`" is introduced both as node and node set variable.

CSR611. Each variable (node or node set) must not occur bounded by more than one nested quantifier (quantifiers are: `ExNode`, `AllNodes`, `ExNodeSet`, `AllNodeSets`, `Child x Con i Node`, *and* `Parent x Con Node` *) in an mso*-formula.*

The definition of quantifier formulas mentioned above allows multi-level of nestedness. With this restriction, it is required that a quantified variable (node or node set) is not re-quantified in the scope of the first quantifier.

CSR612. In an mso-formula may only the variable used in the corresponding left-hand side occur as a free node variable, and moreover, this node variable may not occur bounded in the formula.*

This restriction comes from the requirement that in a definition of an mso*-formula, there is only one free node variable, that is the node argument on the left-hand side. This means there should not be any other free node variable introduced in the right-hand side. Furthermore, any use of the free node variable in the right-hand side is allowed, as long as the free node variable remains free.

CSR613. In a `Label`*-formula, the used constructor must be defined within the input data block and its result type must be an element of the type set which is the declared type of the used node variable.*

A `Label`-formula ("`Label`" `con x`) tests if the node `x` is labeled by the constructor `con`. This restriction requires two conditions:

- `con` is defined in the input data block,

- the result type of `con` is an element of the type set which is declared for the node variable `x`.

CSR614. In an `IsElem`*-formula, the type of the used node variable must be the same as the type of the used node-set variable.*

An `IsElem`-formula (”`IsElem`” `x y`) tests if the node `x` is an element of set of nodes `y`. In order to do that, it requires that the type of `x` is the same as the type of `y`.

CSR615. In an `Equal`*-formula, the types of the two used node variables must be the same.*

An `Equal`-formula (”`Equal`” `x y`) tests if the first node `x` and the second node `y` are equal. In order to do that, it requires that the type of `x` and `y` are the same.

CSR616. In a `TypeOf`*-formula, the used type must be defined within the input data block, and moreover, it must be an element of the type set which is the declared type of the used node variable.*

A `TypeOf`-formula (”`TypeOf`” `x t`) tests if the node `x` has the type `t`. In order to do that, it requires two conditions:

- `t` is defined in the input data block and
- `t` is an element of the type set which is declared for `x`.

CSR617. In an `Edge`*-formula, the used integer value must be greater than zero.*

An `Edge`-formula (”`Edge`” `x i  y`) tests if the node `y` is the `i`-th successor of the node `x`. The integer `i` should be greater than zero which conforms to the fact that a node with successors can only have one or more successor nodes.

CSR618. In a `Mag`*-formula, the used argument must be a well-typed term of type* `Bool` *consisting only of calls of synthesized or inherited functions using node variables as node argument, auxiliary functions, and constructors defined within the input data block.*

A `Mag`-formula (”`Mag`” `RuleRhs`) connects the mso*-logic with attribute grammar formalism. This restriction requires the `RuleRhs` to be a well-typed term of type `Bool`. Well-typeness for `RuleRhs` can be one of the following possibilities:

- it is a synthesized / inherited function which has node variables of the right type as node argument, or
- it is a constructor of type `Bool`.

When `RuleRhs` is an auxiliary function, then we cannot check this.

CSR619. The node argument position of a type assignment must be either the name of a type defined in the input data block or a type variable.

A type assignment can only be found in the following quantifier formulas: `ExNode`, `AllNodes`, `ExNodeSet`, and `AllNodeSets`. This restriction corresponds closely to CSR603, except that this time the restriction is applied to a type assignment, not a type signature.

CSR620. The type variable used in a where-from type restriction must be the same as used in the node argument position in the corresponding type assignment.

This restriction relates to CSR607, in the same manner as CSR619 relates to CSR603.

CSR621. The types used in a where-from type restriction of a type assignment must be defined in the input data block.

This restriction relates to CSR608, in the same manner as CSR619 relates to CSR603.

CSR622. In a where-from type restriction of a type assignment no repetition of type names is allowed.

This restriction relates to CSR609, in the same manner as CSR619 relates to CSR603.

CSR623. In a `Child`*-formula, the used integer value may not be greater than the rank of the used constructor which itself must be defined within the input data block. Moreover, the result type of the used constructor must be an element of the type set which is the declared type of the used node variable (within the* `Child`*-quantifier).*

A `Child`-formula ("`Child`" `x con i Node y` *subformula*) tests if the node `x` is labeled by `con`. Furthermore, in the case where *subformula* holds where `y` is the `i`-th successor node of `x`, then the sort of the node `y` is determined by the sort of `con`. This restriction requires three conditions as follows:

- `i` is smaller or equal to the rank of `con`,
- `con` is defined in the input data block,
- result type of `con` is an element of the type set which is declared for `x`.

CSR624. In a `Parent`*-formula, the used constructor must be defined in the input data block, and moreover, at least one of its argument types must be an element of the type set which is the declared type of the used node variable (within the* `Parent`*-quantifier).*

A `Parent`-formula ("`Parent`" `x con Node y` *subformula*) tests if there exists a parent of the node `x` which is labeled by `con`. Furthermore, in the case where *subformula* holds where `y` is the parent node of `x`, then the sort of the node `y` is determined by the sort of `con`. This restriction requires two conditions as follows:

- `con` is defined in the input data block,
- at least one of the successor types of `con` is an element of the type set which is declared for `x`.

CSR625. Every subformula used within an mso-formula must be defined in the same mso block.*

One of the formula construct in *DSL* is subformula. A formula can invoke other formulas in the form of subformulas. The subformulas invoked are also formulas which should be defined in the same mso block to make it possible to be used by other formulas.

CSR626. The subformula used within another mso-formula must get a node variable as argument such that the type of this node variable and the node argument type of the subformula are the same.*

In accordance with CSR625, this restriction requires that a formula should invoke subformulas by giving the right node variable as argument. This means the type of the given node variable should be the same as the declared node argument type for that formula.

CSR627. No circular dependencies between formulas are allowed in the formula definitions.

The facility to invoke subformulas provides a good structure for clarity and efficiency. On the other hand, it may lead to circular dependencies which will results in infinite invocation of formulas. This restriction requires that we should avoid such circular dependencies between formulas.

CSR628. Every formula which uses node or node set variables must use them correctly. It must use a node variable when a node variable is expected. It must use a node set variable when a node set variable is expected.

A variable can only be introduced as a left-hand side variable or by a quantifier formula. The consequence is:

- There can only be one free node variable, i.e. the left-hand side variable. This is because other variables are introduced by quantifier formulas which make them bounded.

So, this restriction is applied to formulas which "use" a node or a node set variable. Those formulas are `Label`, `IsElem`, `Equal`, `TypeOf`, `Edge`, `Over`, `Under`, `Child`, and `Parent`. (Note that `Child` and `Parent` are also quantifier formulas, but they have the possibility to "use" a node variable in its first variable). When a node / node set variable is used in those formulas, we must make sure that it is the correct node / node set variable. We use the following incorrect example to explain the above restriction:

```
wrong_formula x = ExNode y :: Identifier (IsElem x y)
```

This example is incorrect because the variable `y` is first introduced as a node variable in `ExNode`-formula, but then it is used on the position of a node set variable in `IsElem`-formula.

4 Implementation

In this section, we will discuss about the main work of the project, i.e. the implementation of the syntax-checker for the mso*-logic part of *DSL* specification. This means implementing each of the context-sensitive restrictions that we have discussed in section 3. As implementation language, the functional programming language Haskell [4, 5] is used.

Before we focus on the implementation of the syntax-checker, we need to describe the general framework of the generator, where the syntax-checker resides as a part of the generator. It is given as follows:

1. First of all, a specifier creates a specification of a language L in DSL. Then this specification is given to the generator as input.

2. The generator scans and parses the specification using its scanner and parser. The parser transforms the specification into a tree structure defined by data types in Haskell.

3. The syntax-checker, which is a part of the generator, will take the following parsed part of the specification as its input:

   - the mso*-logic part
   - the abstract syntax part
   - the function environment part, which includes the attribution, unparsing, and structural transformations structures of the specification.

4. The syntax-checker will collect information from the above input items and check the information against the context-sensitive restrictions. As a result, it will give error messages if the specification violates the context-sensitive restrictions. Otherwise the generator will continue to generate a language-based editor for L.

In order to discuss the implementation task, we need to introduce the internal data types used by the parser. This is important regarding the fact that the syntax-checker works based on these internal data types. After that, we explain how we implement the syntax-checker. We divide the syntax-checker into three part, viz checker for general restrictions, checker for restrictions on formula declarations, and checker for restrictions on formula definitions, just as we classify those context-sensitive restrictions in section 3. Then we discuss about the auxiliary functions which are frequently used in implementing the restrictions. As the last part, we explain about the implementation of the user interface.

4.1 Internal Data Types from the Parser

Now we will discuss about the internal data types used by the parser. These data types are defined using Haskell. Basically, these data types represent DSL non-terminal symbols. They cover the definition of mso block, data block and function environment of L specification.

Before we explain about the data types which represent some of the DSL symbols, we will give the following type renamings for `String` to make things more clear:

```
type ConName  = String
type TypeName = String
type VarName =  String
type Var1Name =  String
type Var2Name =  String
type FormulaName = String
type DataDefName = String
type InputDefName  =  String
```

`ConName` stores constructor name, `TypeName` stores type name, `VarName` stores variable (node or node set), `Var1Name` stores node variable, `Var2Name` stores node set variable, `FormulaName` stores formula name, `DataDefName` stores data block name, and `InputDefName` stores input data block name.

Now we start from the data types `MsoBlock`, `InputDef` and `MsoDefList`. `MsoBlock`, together with `MsoDefList`, represents the DSL symbol $\langle MsoBlock \rangle$. This DSL symbol is the start symbol for an mso-logic part of DSL specification. While `InputDef` represents the symbol $\langle InputDef \rangle$. Here are their definitions:

```
data MsoBlock
  = MsoBlock String InputDef MsoDefList

data InputDef
  = InputDef String

data MsoDefList
  = MsoDefListSingle MsoDef
  | MsoDefListPair MsoDefList MsoDef
```

`MsoBlock` defines the whole structure of an mso block. It has one constructor `MsoBlock`, followed by a `String`, `InputDef`, and `MsoDefList`. The `String` stores the name of the mso block. `InputDef` stores the name of the input data block. While `MsoDefList` stores a list of formula declarations and / or definitions, by using the data type `MsoDef`. From the way `MsoDefList` is defined, we guarantee that an mso block consists of at least one formula declaration or one formula definition. `MsoDefList` has two constructors: `MsoDefListSingle` and `MsoDefListPair`. By using `MsoDefListSingle`, we create a single formula declaration / definition. While by using `MsoDefListPair`, we have the ability to create a list of formula declarations and / or definitions.

Here we explain about `MsoDef` which is used in the definition of `MsoDefList`. It represents $\langle MsoDef \rangle$ and is defined as follows:

```
data MsoDef
  = MsoDefType TypeDef
  | MsoDefFormula FormulaDef
```

`MsoDef` has two constructors: `MsoDefType` and `MsoDefFormula`. `MsoDefType` is used to define a formula declaration, using its successor type `TypeDef`. `MsoDefFormula` is used to define a formula definition, using its successor type `FormulaDef`.

`TypeDef` represents $\langle MsoTypeDef \rangle$ and is defined as follows:

```
data TypeDef
  = TypeDef AttrType String TypeSigOpt
```

`TypeDef` stores a formula declaration. A formula declaration should consist of a formula name and a type signature. This data type has one constructor `TypeDef` with three successor types: `AttrType`, `String`, and `TypeSigOpt`. Here is the definition of `AttrType`:

```
data AttrType
  = Syn
  | Inh
```

```
    | Unparse
    | Transform
    | Formula
```

`AttrType` provides options for the types of attribute, such as `Syn`, `Inh`, `Unparse`, `Transform`, and `Formula`. In practice, the parser will always set the attribute type to `Formula` in an mso block definition. Other attributes may be used in function environment. In DSL symbol $\langle MsoTypeDef \rangle$, the attribute type is set to the terminal symbol "`formula`".

The second successor type `String` of `TypeDef` stores the name of the formula. While the third successor type `TypeSigOpt` is defined as follows:

```
data TypeSigOpt
  = TypeSigOptNull
  | TypeSigOpt TypeSig
```

`TypeSigOpt` defines the type signature of the formula. It has constructors `TypeSigOptNull` and `TypeSigOpt`. `TypeSigOptNull` is a nullary constructor which defines an empty type signature. While `TypeSigOpt` is a unary constructor which defines a non-empty type signature, using the successor type `TypeSig`. We will explain about `TypeSig` later when we discuss about the DSL symbol $\langle TypeSig \rangle$.

`FormulaDef` represents $\langle FormulaDef \rangle$ and is defined as follows:

```
data FormulaDef
  = FormulaDef FormulaName Var1Name FoStruc_Rs
```

`FormulaDef` stores a formula definition. A formula definition should consist of a left-hand and a right-hand side. It has one constructor `FormulaDef` with successor types: `FormulaName`, `Var1Name`, and `FoStruc_Rs`. `FormulaName` and `Var1Name` are used on the left-hand side. They specify the formula name and the only free node variable in that formula. While `FoStruc_Rs` is used on the right-hand side to define the node property.

`FoStruc_Rs` represents $\langle Formula \rangle$ which is also defined by other DSL symbols such as $\langle boolconst \rangle$, $\langle Base \rangle$, $\langle Connection \rangle$, and $\langle Quantification \rangle$. Here is the definition of `FoStruc_Rs`:

```
data FoStruc_Rs
  = FS_CONSTANT Bool
  | FS_LABEL ConName Var1Name
  | FS_ELEM Var1Name Var2Name
  | FS_EQUAL Var1Name Var2Name
  | FS_TYPEOF Var1Name TypeName
  | FS_EDGE Var1Name Int Var1Name
  | FS_OVER Var1Name Var1Name
  | FS_UNDER Var1Name Var1Name
  | FS_SUB FormulaName Var1Name
  | FS_PRED RuleRhs
  | FS_NEG FoStruc_Rs
  | FS_AND FoStruc_Rs FoStruc_Rs
  | FS_EXISTS1 (Var1Name,UpperIdList,(String,String)) FoStruc_Rs
```

```
| FS_EXISTS2 (Var2Name,UpperIdList,(String,String)) FoStruc_Rs
| FS_ALL1 (Var1Name,UpperIdList,(String,String)) FoStruc_Rs
| FS_ALL2 (Var2Name,UpperIdList,(String,String)) FoStruc_Rs
| FS_CHILD Var1Name ConName Int Var1Name FoStruc_Rs
| FS_PARENT Var1Name ConName Var1Name FoStruc_Rs
```

`FoStruc_Rs` defines the structure of the right-hand side of an mso*-formula. We will explain below in more detail about the constructors of `FoStruc_Rs`:

- `FS_CONSTANT` constructs a constant formula which relates with ⟨*boolconst*⟩.
- `FS_LABEL`, `FS_ELEM`, `FS_EQUAL`, `FS_TYPEOF`, `FS_EDGE`, `FS_OVER`, and `FS_UNDER` construct basic formulas which relate with ⟨*Base*⟩.
- `FS_SUB` allows us to reuse a defined formula as a subformula of a larger formula. In DSL, this subformula is represented by two subsequent non-terminal symbols: ⟨*questid*⟩ ⟨*lowerid*⟩.
- `FS_PRED` constructs an attribute formula which calls attributes defined in the attribution structure of DSL specification. In DSL syntax, this formula is represented by a combination of terminal and non-terminal symbols: "`Mag`" "`(`" ⟨*RuleRhs*⟩ "`)`".
- `FS_NEG` and `FS_AND` construct connection formulas which relate with ⟨*Connection*⟩.
- `FS_EXISTS1`, `FS_EXISTS2`, `FS_ALL1`, `FS_ALL2`, `FS_CHILD`, and `FS_PARENT` construct quantifier formulas which relate with ⟨*Quantification*⟩. Their first successor type represents a type assignment which is denoted by ⟨*TypeAss*⟩ in DSL. Type assignment can only be found in those four quantifier formulas. In the parser, type assignment is a tuple containing three elements as follows:
  - `Var1Name` or `Var2Name`: stores the quantified variable (node or node set),
  - `UpperIdlist`: stores the types of the quantified variable,
  - a tuple (`String`, `String`): stores the type variable and the type restriction variable. When no type variable is used in the type assignment, then both `String` are set to empty `String`: "".

  The second successor type of the quantifier formulas is again `FoStruc_Rs` which relates with ⟨*ClasFor*⟩.

  Here are two examples of `ExNode`-formula in DSL syntax and in the parser's data type: the first one doesn't use any type variable, and the second one uses a type variable:

  1. Without type variable:
     - In DSL syntax:
       ```
       ExNode y :: Identifier (?example_formula y)
       ```
     - In parser's data type:
       ```
       FS_EXISTS1 ("y", UpperIdListSingle "Identifier", ("",""))
                  (FS_SUB "example_formula" "y")
       ```

2. With type variable:
   - In DSL syntax:
     ```
     ExNode y :: a where a from {Identifier, Expression}
       (?example_formula y)
     ```
   - In parser's data type:
     ```
     FS_EXISTS1 ("y",
                 UpperIdListPair
                   (UpperIdListSingle "Identifier")
                   "Expression",
                 ("a","a"))
       (FS_SUB "example_formula" "y")
     ```

Now we will discuss about `TypeSig` which is used in the definition of `TypeSigOpt` above. `TypeSig`, together with the following data types: `TreeArgType`, `TypeArrowList`, `TransResultType`, `TypeRestrOpt`, `TypeRestr`, and `UpperIdList`, represents the DSL symbol $\langle TypeSig \rangle$. Those data types are defined as follows:

```
data TypeSig
  = TypeSig TreeArgType TypeArrowList TransResultType TypeRestrOpt

data TreeArgType
  = TreeArgTypeUpperId String
  | TreeArgTypeLowerId String

data TypeArrowList
  = TypeArrowListSingle Type
  | TypeArrowListPair TypeArrowList Type

data TransResultType
  = TransResultTypeNull
  | TransResultType String

data TypeRestrOpt
  = TypeRestrOptNull
  | TypeRestrOpt TypeRestr

data TypeRestr
  = TypeRestr String UpperIdList

data UpperIdList
  = UpperIdListSingle String
  | UpperIdListPair UpperIdList String
```

`TypeSig` has only one constructor of the same name, i.e. `TypeSig`. This data type defines the structure of a type signature. The constructor `TypeSig` has four successor types as follows:

- `TreeArgType`: defines the node argument type,
- `TypeArrowList`: defines the types of the context parameters,
- `TransResultType`: defines the result type of the formula,
- `TypeRestrOpt`: defines the where-from type restriction.

`TreeArgType` has two constructors: `TreeArgTypeUpperId` and `TreeArgTypeLowerId`. Both have `String` as their only successor type. The first constructor is used to construct a node argument type in the form of a type defined in the input data block. While the second one is used to construct a node argument type in the form of a type variable.

`TypeArrowList` has two constructors: `TypeArrowListSingle` and `TypeArrowListPair`. The first constructor is used to construct a single context parameter type. While the second one is used to construct more than one context parameter type. They both use `Type` which we will explain later.

`TransResultType` has two constructors: `TransResultTypeNull` and `TransResultType`. The first constructor constructs an empty result type. While the second one constructs a non-empty result type, using the successor type `String`.

`TypeRestrOpt` has two constructors:`TypeRestrOptNull` and `TypeRestrOpt`. The first constructor constructs an empty where-from type restriction. While the second one constructs a non-empty where-from type restriction, using the successor type `TypeRestr`.

`TypeRestr` has only one constructor of the same name, i.e. `TypeRestr`. This data type defines the structure of a where-from type restriction. The constructor `TypeRestr` has two successor types as follows:

- `String`: stores the type restriction variable, and
- `UpperIdList`: defines the types used in a where-from type restriction. Those types should be defined in the input data block.

Now we will explain about `Type` which is used in the definition of `TypeArrowList` above. The data type `Type`, together with `TypeCommaList`, represents the DSL symbol $\langle Type \rangle$. Here are the definitions of `Type` and `TypeCommaList`:

```
data Type
  = SimpleType String
  | ListType Type
  | TupleType Type TypeCommaList

data TypeCommaList
  = TypeCommaListSingle Type
  | TypeCommaListPair TypeCommaList Type
```

`Type` has three constructors: `SimpleType`, `ListType`, and `TupleType`. `SimpleType` constructs a single type. `ListType` constructs a list of types. While `TupleType` constructs a tuple of types, using `Type` and `TypeCommaList`.

The data type `RuleRhs`, together with `RhsExpList0`, `RhsExpList`, `RhsExp`, `RhsInfixOp`, `RhsExpInfixOpList`, `Const`, `Compound`, and `RuleRhsCommaList`, defines the general structure of a right-hand side of a macro attribute definition. It is used in an attribute formula which connects the mso*-logic with the macro attribute formalism. Their correlation with DSL definition are explained as follows:

- RuleRhs correlates wih $\langle RuleRhs \rangle$,
- RhsExpList0, RhsExpList, and RhsExp correlate with $\langle RhsExp \rangle$,
- RhsInfixOp and RhsExpInfixOpList correlate with $\langle InfixOp \rangle$,
- Const correlates with $\langle Const \rangle$, and
- Compound and RuleRhsCommaList correlate with $\langle Compound \rangle$.

Here are the definitions of the above mentioned data types:

```
data RuleRhs
  = RuleRhsLowerIdpp String RhsExpList0
  | RuleRhsUpperIdpp String RhsExpList0
  | RuleRhsInfixOp RhsExp RhsInfixOp RhsExpInfixOpList
  | RuleRhsPreInfixOp String RhsExpInfixOpList
  | RuleRhsConst Const
  | RuleRhsCompound Compound

data RhsExpList0
  =  RhsExpList0Empty
  | RhsExpList0List RhsExpList

data RhsExpList
  = RhsExpListSingle RhsExp
  | RhsExpListPair RhsExpList RhsExp

data RhsExp
  = RhsExpLowerId String
  | RhsExpUpperId String
  | RhsExpConst Const
  | RhsExpCompound Compound

data RhsInfixOp
  = RhsInfixOp String
  | RhsPreInfixOp String
```

```
data RhsExpInfixOpList
  = RhsExpInfixOpListSingle RhsExp
  | RhsExpInfixOpListPair RhsExpInfixOpList RhsInfixOp RhsExp

data Const
  = ConstBool Bool
  | ConstInt Int
  | ConstChar String
  | ConstString String
  | ConstEmptyList

data Compound
  = CompoundTuple RuleRhsCommaList
  | CompoundList RuleRhsCommaList

data RuleRhsCommaList
  = RuleRhsCommaListSingle RuleRhs
  | RuleRhsCommaListPair RuleRhsCommaList RuleRhs
```

The following data types corresponds with the data block definition of DSL. The correlation is explained as follows:

- DataDefList and DataDef correlate with $\langle DataDef \rangle$,
- ConsList and Cons correlate with $\langle Cons \rangle$, and
- TypeList0 correlates to a list of $\langle Type \rangle$.

DataDefList, DataDef, ConsList, Cons and TypeList0 are defined as follows:

```
data DataDefList
  = DataDefListSingle DataDef
  | DataDefListPair DataDefList DataDef

data DataDef
  = DataDef String ConsList

data ConsList
  = ConsListSingle Cons
  | ConsListPair ConsList Cons

data Cons
  = Cons String TypeList0

data TypeList0
  = TypeList0Empty
  | TypeList0List TypeList
```

```
data TypeList
  = TypeListSingle Type
  | TypeListPair TypeList Type
```

The following type renamings correspond to information about data blocks and function environment. `DataEnv` represents $\langle DataBlock \rangle$ and defines a collection of data blocks. `FunEnv` represents the function environment collection of macro attribute grammar, unparsing part and structural transformations . Each FunTab represent a collection of function for each environment, distinguished by its name.

```
type DataEnv = [(String,DataDefList)]
type FunEnv  = [(String,FunTab)]
type FunTab  = [(AttrType,String,TypeSigOpt)]
```

4.2 Implementation of the Context-Sensitive Restrictions

As explained in section 3, there are twenty-eight context-sensitive restrictions for mso*-logic part of DSL specification. We implement each of these restrictions against the parsed specification. This parsed specification is divided into three parts: mso*-logic, abstract syntax, and function environment. Those three parts act as input for the syntax-checker of mso*-logic. But only CSR618 which will use the information from the function environment part. The parser uses the data types which we have explained in the previous subsection.

If a restriction is violated, we store the error information in a data type `ErrDat`. `ErrDat` has a unique constructor name for each restriction. It stores different information based on each of the restrictions. We will explain more about the error information when we discuss the implementation of each restriction. Here is an illustration of how `ErrDat` is defined:

```
data ErrDat
  = ErrDatCSR601 ...
  | ErrDatCSR602 ...
  | ...
  | ErrDatCSR628 ...
```

In this implementation, we also need to create several auxiliary functions. This is necessary because they are designed to do some general tasks which are found in some restrictions. As a result, we will improve modularity and efficiency. We will discuss more about those auxiliary functions in the next subsection.

Below we divide the restrictions into three part: general restrictions, restrictions on formula declarations, and restrictions on formula definitions.

4.2.1 Implementation of the General Restrictions

In this part, we will realize the checking for CSR601 and CSR602. CSR601 will need information from mso block and input data block. While CSR602 will need information only from mso block.

CSR601

Here is the general idea to make sure that the input definition of mso block uses a defined data block:

1. extract the input definition name from mso block:

   `MsoBlock` $\leadsto$ `InputDef` $\leadsto$ *input definition name*,

2. collect a list of data block names from `DataEnv`,

3. check whether we can find the input definition name of mso block in the list of data block names. If we find it, then CSR601 is fulfilled. Otherwise, CSR601 is violated, and we return the undefined input definition name as the error data:

   ```
   ErrDatCSR601 InputDefName
   ```

CSR602

Every formula defined in mso block should be declared first within the same mso block. This is done as follows:

- collect all defined formula names:

  `MsoBlock` $\leadsto$ *a list of* `FormulaDef` $\leadsto$ *a list of defined formula names*

  We use function *getForDef* to get a list of `FormulaDef` from `MsoBlock`.

- Collect all declared formula names:

  `MsoBlock` $\leadsto$ *a list of* `TypeDef` $\leadsto$ *a list of declared formula name*

  We use function *getTypeDef* to get a list of `TypeDef` from `MsoBlock`.

- checking whether every defined formula name can be found in the list of declared formula names. If it is so, then CSR602 is fulfilled. Otherwise, CSR602 is violated, and we return the list of undeclared formula names as the error data:

  ```
  ErrDatCSR602 [FormulaName]
  ```

4.2.2 Implementation of Restrictions on Formula Declarations

In this part, we will realize the checking for CSR603 until CSR609. These restrictions deal mainly with the formula declarations part of mso block, i.e. `TypeDef`. This will make mso block as the main source of input for this implementation. But since formula declarations consists of type signatures, then we will also need information from input data block, which defines the types in a language.

CSR603

According to this restriction, the node argument position of a type signature must be one of the following possibilities: it is the name of a type defined in the input data block, or it is a type variable. We do the following steps to check this property:

- extract the node argument information of each formula. The information should consist of:

  1. the formula name,
  2. its node argument type variable, if there is any,
  3. the list of node argument types,
  4. the where-from type restriction, if there is any.

  We use function *getForNodeArg* to get the information above.

- Collect all types defined in the input data block. We use function *getAllTypes* which collects all the types from input data block given by the function *getInputDataBlock*.

- If the node argument position is not a type variable, then we check whether that type is an element of the list of types defined in the input data block. If it is, then CSR603 is fulfilled. Otherwise, CSR603 is violated, and we return the formula name and its undefined node argument type as error data:

  ```
  ErrDatCSR603 FormulaName TypeName
  ```

  If the node argument position is a type variable, here we do not care further and we say CSR603 is fulfilled.

CSR604

In the type signature of mso*-formula, only `Bool` is allowed as result type. This can be done by taking the type signature and then examining only the last argument type of the type signature. If the last argument of the type signature is `Bool`, then CSR604 is fulfilled. If not, then we return error data containing the formula name and the wrong result type:

```
ErrDatCSR604 FormulaName Type
```

CSR605

The type signature of mso*-formula must not define context parameters. This restriction can be checked by extracting the type signature of each formula declarations in mso block:

`MsoBlock` ⇝ `TypeDef` ⇝ `TypeSigOpt` ⇝ `TypeSig`

Here we use again the function *getTypeDef* to get a a list of `TypeDef`.

Now we check the type signature. It should consist of only two types: a node argument type and a result type. If there are more types between the node argument and the result type, then those types are the context parameters. In that case, CSR605 is violated, and we return the list of formula names which violate CSR605 as error data:

```
ErrDatCSR605 [FormulaName]
```

CSR606

A where-from type restriction must be used if and only if a type variable is used in the node argument position of a type signature. Here we need to use function *getForNodeArg* again, to get information on node argument of each formula declaration. Given a list of information about node argument of each formula, we do the following:

- filter the list to contain only formulas having type variable as node argument type,
- check whether each of the above formulas has a where-from type restriction. If it does, then CSR606 is fulfilled. Otherwise, CSR606 is violated, and we return the list of formula names which violate CSR606 as error data:

```
ErrDatCSR606 [FormulaName]
```

CSR607

The type variable used in a where-from type restriction must be the same as used in the node argument position in the corresponding type signature. The first two steps are exactly the same as what we do in CSR606. But when the formula contains a where-from type restriction, we do one step more. This step is to check whether type variable used in the where-from-type restriction is the same as the the type variable in the node argument. If it is, then CSR607 is fulfilled. Otherwise, CSR607 is violated, and we return the formula name, the type variable on node argument position, and the incorrect type variable on the where-from type restriction as error data:

```
ErrDatCSR607 FormulaName Var1Name Var1Name
```

CSR608

The types used in a where-from type restriction must be defined in the input data block. Just like what we do in CSR607, here we follow the same two steps from CSR606. But when the formula contains a where-from type restriction, we do one step more. This step is to check whether the types used in the where-from type restriction are defined in the input data block. We use again the function *getAllTypes* and *getInputDataBlock* to collect all the types defined in the input data block.

If all the types in the where-from type restriction are defined in the input data block, then CSR608 is fulfilled. Otherwise, it is violated, and we return the formula name and its undefined types in the where-from type restriction as error data:

```
ErrDatCSR608 FormulaName [TypeName]
```

CSR609

This restriction relates closely with CSR608, in the sense that it also concerns about the types used in a where-from restriction. This time, the restriction requires that the types are not repeated. We do just exactly as in CSR608, except here we check whether there is repetition of types in the where-from type restriction. If there is no repetition, then CSR609 is fulfilled. Otherwise, it is violated, and we return the formula name and its repeated restriction types as error data:

```
ErrDatCSR609 FormulaName [TypeName]
```

4.2.3 Implementation of Restrictions on Formula Definitions

Now we will discuss about the implementation of CSR610 until CSR628. These restrictions concern with the contents of formula definitions. So, all of them need input information from mso block. Information from input data block is also needed when we want to check whether a type used in the formula definition is defined. CSR618 is the only restriction which needs input from function environment, beside from mso block and input data block.

In general, the main characteristic of these restrictions is that they all need to use function *getForInfo*. This function traverses down the tree structure of formula definitions and returns a well-structured information about each of them. We will call the information as *formula definition information*. Using this information, each restriction function can easily check the validity of formula definitions. A formula definition information consists of formula name, node argument variable, the types of the node argument variable, and a list of *formula structures*. A formula structure basically stores the categorization of variables in each subformula structure into free node variables, bounded node variables, free node set variables, and bounded node set variables. The list of formula structures is given by function *getStruc*. We will discuss more about functions *getForInfo* and *getStruc*, and the data types of formula definition information and formula structure in the next subsection.

CSR610

A variable cannot be a node variable and a node set variable in the same mso*-formula. To check this property, we do the following steps:

- call the function *getForInfo*,
- collect all node variables found in each formula definition. We can collect the node variables easily from the list of formula structures in formula definition information given by *getForInfo*. The collection of node variables should consist of:
  - free node variables,
  - bounded node variables, and
  - the free node variable on the left-hand side of formula definition.
- Collect all node set variables found in each formula definition. They consist of:
  - free node set variables and
  - bounded node set variables
- Check whether the collection of node variables and node set variables share some equal elements. If they don't, then CSR610 is fulfilled. Otherwise, it is violated, and we return the formula name and a list of variables found in both collection as error data:

```
ErrDatCSR610 FormulaName [VarName]
```

CSR611

Each variable (node or node set) must not occur bounded by more than one nested quantifier in a formula. Quantifier formulas are the ones constructed by the following constructors of `FoStruc_Rs`:

1. `FS_EXISTS1`,
2. `FS_ALL1`,
3. `FS_EXISTS2`,
4. `FS_ALL2`,
5. `FS_CHILD`,
6. `FS_PARENT`,

Here are the steps to check the above property:

- call the function *getForInfo*,
- check every subformula structure which uses one of the constructors above. We must check whether they quantify a variable which is already quantified in the scope of an outer quantifier. This checking is done for every quantifier formula. If the quantified variables are not previously quantified, then CSR611 is fulfilled. Otherwise, it is violated, and we return the formula name and a list of data type `VarNestedBounded` as error data:

  ```
  ErrDatCSR611 FormulaName [VarNestedBounded]
  ```

  `VarNestedBounded` stores information about a nested-quantified variable in the subformula structures. The definition of `VarNestedBounded` is given as follows:

  ```
  data VarNestedBounded
    = VarNestedBounded FoStruc_Rs Var1Name Var2Name
  ```

  `VarNestedBounded` consists of a constructor `VarNestedBounded`, followed by:

  - `FoStruc_Rs`: stores the current subformula structure,
  - `Var1Name`: stores the name of a node variable which is quantified in the current subformula but has been previously quantified in an outer subformula,
  - `Var2Name`: stores the name of a node set variable which is quantified in the current subformula but has been previously quantified in an outer subformula.

CSR612

In an mso*-formula, only the variable used in the corresponding left-hand side may occur as a free node variable. Moreover, this node variable may not occur bounded in the formula. This is done as follows:

- extract formula definition information, using function *getForInfo*,

- check whether there are other free node variables introduced in each formula definition. This is done by collecting all free node variables from formula definition information. If there are some free node variables other than the left-hand side variable, then we store it in the list of `ForWithNonLhsFreeNV`.
- Check whether there are subformulas in each formula definition which bound the left-hand side variable. We store the formulas which incorrectly bound the left-hand side variable as a list of `FoStruc_Rs`.
- Decide whether a formula definition violates CSR612 or not. We do this by examining the list of `ForWithNonLhsFreeNV` and the list of `FoStruc_Rs` which are constructed in the two steps above. If both lists are empty, then CSR612 is completely fulfilled. Otherwise, CSR612 is violated, and we return the formula name and those two lists as error data:

  ```
  ErrDatCSR612 FormulaName [ForWithNonLhsFreeNV] [FoStruc_Rs]
  ```

  Here is the definition of `ForWithNonLhsFreeNV`:

  ```
  data ForWithNonLhsFreeNV
    = ForWithNonLhsFreeNV FoStruc_Rs [Var1Name]
  ```

  `ForWithNonLhsFreeNV` stores the invalid subformula structure and the list of illegal free node / node set variable which is different from the left-hand side variable.

CSR613

A `Label`-formula must fulfill the following conditions:

- the used constructor must be defined within the input data block, and
- the constructor's result type must be an element of the type set which is declared for the used node variable.

To check this property, we do as follows :

- extract formula definition information, using *getForInfo*,
- filter the formula structures to contain only `Label`-formula as subformulas,
- check each `Label`-formula in the filtered list of formula structures. We create the data type `IncorrectLabel` to store error information of each subformula structure. It is defined as follows:

  ```
  data IncorrectLabel
    = IncorrectLabel FoStruc_Rs TypeName [TypeName]
  ```

  `IncorrectLabel` stores the incorrect `Label`-formula, the result type of the used constructor, and the types declared for the used node variable.

  If the constructor name used in the `Label`-formula is defined in the input data block, then we go on by checking whether the the result type of the constructor is an element of the type declared for the used node variable. If the result type of the constructor is not an element of the type declared for the used node variable, then we store that error in the data type `IncorrectLabel`, containing the `Label`-formula, the result type of the constructor, and the declared type for the used node variable.

If the used constructor name is not defined, then further checking regarding the result type of the constructor is not needed. We store the error in `IncorrectLabel`, by setting the value of the type `TypeName` as empty string and the type `[TypeName]` as empty list.

- Create error data based on the list of `IncorrectLabel`. If that list is empty, then the formula definition does not violate CSR613. Otherwise, we return the formula name and the list of `IncorrectLabel` as error data:

```
ErrDatCSR613 FormulaName [IncorrectLabel]
```

CSR614

In an `IsElem`-formula, the type of the used node variable must be the same as the type of the used node-set variable. Here are the steps to check this property:

- extract formula definition information, using *getForInfo*,
- filter the formula structures to contain only `IsElem`-formula as subformulas,
- check each `IsElem`-formula in the filtered list of formula structures. We do this by examining the types of node variable and node set variable in `IsElem`-formula. Both types should be the same. If their types are different, then we store the error in data type `IncorrectElem`. It is defined as follows:

```
IncorrectElem FoStruc_Rs [TypeName] [TypeName]
```

  `IncorrectElem` stores the incorrect `IsElem`-formula, the types of the node variable, and the types of the node set variable.

- Create error data based on the list of `IncorrectElem`. If that list is empty, then the formula definition does not violate CSR614. Otherwise, we return the formula name and the list of `IncorrectElem` as error data:

```
ErrDatCSR614 FormulaName [IncorrectElem]
```

CSR615

In an `Equal`-formula, the types of the two used node variables are the same. To check this property, we do as follows:

- extract formula definition information, using *getForInfo*,
- filter the formula structures to contain only `Equal`-formula as subformulas,
- check each `Equal`-formula in the filtered list of formula structures. The two used node variables in `Equal`-formula should have the same type. If their types are different, then we store the error in data type `IncorrectEqual`, which is defined as follows:

```
data IncorrectEqual
  = IncorrectEqual FoStruc_Rs [TypeName] [TypeName]
```

`IncorrectEqual` stores the incorrect `Equal`-formula, the types of the first node variable, and the types of the second node variable.

- Create error data based on the list of `IncorrectEqual`. If that list is empty, then the formula definition does not violate CSR615. Otherwise, we return the formula name and the list of `IncorrectEqual` as error data:

```
ErrDatCSR615 FormulaName [IncorrectEqual]
```

CSR616

In a `TypeOf`-formula, the used type must be defined within the input data block, and it is an element of the declared type set for the used node variable. This checking is done as follows:

- extract formula definition information, using *getForInfo*,
- filter the formula structures to contain only `TypeOf`-formula as subformulas,
- check each `TypeOf`-formula in the filtered list of formula structures. We store the error information of each incorrect `TypeOf`-formula in a data type `IncorrectTypeOf`. It is defined as follows:

```
data IncorrectTypeOf
  = IncorrectTypeOf FoStruc_Rs Int [TypeName]
```

`IncorrectTypeOf` stores the incorrect `TypeOf`-formula, an integer differenciating the possible error cases, and the types declared for the used node variable.

If the used type is defined in the input data block, then we go on by checking whether it is an element of the type declared for the used node variable in `TypeOf`-formula. If it is not an element of the type declared for the used node variable, then we store that error in the data type `IncorrectTypeOf`, by setting the integer value to 2.

If the used type is not defined in the input data block, then we store the error in `IncorrectTypeOf`, by setting the integer value to 1. Further checking is not needed since the type itself is not defined in the input data block.

- Create error data based on the list of `IncorrectTypeOf`. If that list is empty, then this formula definition does not violate CSR616. Otherwise, we return the formula name and the list of `IncorrectTypeOf` as error data:

```
ErrDatCSR616 FormulaName [IncorrectTypeOf]
```

CSR617

In an `Edge`-formula, the used integer value must be greater than zero. This checking is done as follows:

- extract formula definition information, using *getForInfo*,

- filter the formula structures to contain only `Edge`-formula as subformulas,
- check each `Edge`-formula in the filtered list of formula structures. If the integer number in `Edge`-formula is less than or equal zero, then it is a wrong formula and we store the incorrect formula into a list of `FoStruc_Rs`.
- If for each formula definition, the list of incorrect `Edge`-formulas is empty, then this formula definition does not violate CSR617. Otherwise, we return the formula name and the list of incorrect `Edge`-formulas as error data:

```
ErrDatCSR617 FormulaName [FoStruc_Rs]
```

CSR618
In a `Mag`-formula, the used argument must be a well-typed term of type Bool consisting only of calls of synthesized or inherited functions using node variables as node argument, auxiliary functions, and constructors defined within the input data block. This checking is done as follows:

- extract formula definition information, using *getForInfo*,
- filter the formula structures to contain only `Mag`-formula as subformulas,
- check the correctness of each `Mag`-formula in the filtered list. The synthesized / inherited function used in `RuleRhs` of `Mag`-formula should be defined in the function environment. Furthermore, those synthesized / inherited functions should be given a node argument variable with the right type, as it is defined in function environment. If the `Mag`-formula is incorrect, we collect that formula into a list of `FoStruc_Rs`.
- If for each formula definition, the list of incorrect `Mag`-formula is empty, then this definition does not violate CSR618. Otherwise, we return the formula name and the list of incorrect `Mag`-formula as error data:

```
ErrDatCSR618 FormulaName [FoStruc_Rs]
```

CSR619.
The node argument position of a type assignment must be either the name of a type defined in the input data block or a type variable. To check this property, we do as follows:

- extract formula definition information, using *getForInfo*,
- filter the formula structures to contain only subformulas which introduce type assignment. There are four formula constructs which introduce type assignment, viz:
  - `FS_EXISTS1` *type_assignment subformula*,
  - `FS_EXISTS2` *type_assignment subformula*,
  - `FS_ALL1` *type_assignment subformula*, and
  - `FS_ALL2` *type_assignment subformula*.

In the parser, type assignment is represented as follows:

```
(VarName,UpperIdList,(String,String))
```

If the value of (`String, String`) is equal to ("", ""), then it means that the node argument of the type assignment does not use a type variable. Otherwise, it uses a type variable.

- Collect all types defined in the input data block, using *getAllTypes*.
- If the node argument does not use a type variable, then we check whether the type used is an element of the list of types defined in the input data block. If the type is defined, then CSR619 is fulfilled. Otherwise, it is violated, and we return the formula name and the list of incorrect formulas as error data:

  ```
  ErrDatCSR619 FormulaName [FoStruc_Rs]
  ```

  If the node argument position is a type variable, then we do not care further and we say CSR619 is fulfilled.

CSR620

The type variable used in a where-from type restriction must be the same as used in the node argument position in the corresponding type assignment. To check this property, we do as follows:

- do the first two steps as explained in CSR619. If the value of (`String, String`) is not equal to ("",""), then it represents the node argument type variable and the type restriction variable of the type assignment.
- Check whether the type restriction variable in the where-from-type restriction of a type assignment is the same as the the type variable used in the node argument position of the corresponding type assignment. This means checking the two `String` values in (`String, String`). If they are the same, then CSR620 is fulfilled. Otherwise, CSR620 is violated and we return the formula name and the list of incorrect formulas as error data:

  ```
  ErrDatCSR620 FormulaName [FoStruc_Rs]
  ```

CSR621

The types used in a where-from type restriction of a type assignment must be defined in the input data block. To check this property, we do as follows:

- do the first three steps as in CSR619,
- check whether the types used in the where-from-type restriction of a type assignment is an element of the list of types defined in the input data block. If they are defined, then CSR621 is fulfilled. Otherwise, CSR621 is violated and we return the formula name and a list of `TypeRestrTypeAssNotDefined` as error data:

  ```
  ErrDatCSR621 FormulaName [TypeRestrTypeAssNotDefined]
  ```

`TypeRestrTypeAssNotDefined` contains the incorrect formula and the list of undefined types in its where-from type restriction. It is defined as follows:

```
data TypeRestrTypeAssNotDefined
  = TypeRestrTypeAssNotDefined FoStruc_Rs [TypeName]
```

CSR622

In a where-from type restriction of a type assignment no repetition of type names is allowed. To check this property, we do as follows:

- do the first two steps as explained in CSR619,
- check whether the types used in the where-from-type restriction of a type assignment is not repeated. If they are not, then CSR621 is fulfilled. Otherwise, CSR621 is violated and we return the formula name and a list of `TypeRestrTypeAssRepeated` as error data:

  ```
  ErrDatCSR621 FormulaName [TypeRestrTypeAssRepeated]
  ```

  `TypeRestrTypeAssRepeated` contains the incorrect formula and the list of repeated types in its where-from type restriction. It is defined as follows:

  ```
  data TypeRestrTypeAssRepeated
    = TypeRestrTypeAssRepeated FoStruc_Rs [TypeName]
  ```

CSR623

A `Child`-formula must fulfill the following conditions:

1. the used constructor is defined within the input data block,
2. the used integer value is greater than the rank of the used constructor, and
3. the result type of the used constructor is an element of the type set which is the declared type of the used node variable.

To check the above properties, we do as follows:

- extract formula definition information, using *getForInfo*,
- filter the formula structures to contain only `Child`-formula as subformulas,
- check the above three restrictions in the following manner: first we check whether the constructor is defined in the input data block. If it is not defined, then there is no need to check the other two conditions. Otherwise, we continue by checking the other two conditions independently. We define the data type `IncorrectChild` to store information about the wrong `Child`-formula:

  ```
  data IncorrectChild
    = IncorrectChild FoStruc_Rs TypeName Int [TypeName]
  ```

  It consists of the following elements:

- the constructor: `IncorrectChild`,
- `FoStruc_Rs`: the wrong `Child`-formula,
- `TypeName`: the result type of the used constructor in the `Child`-formula. It is given value empty `String` when the constructor is not defined in the input data block,
- `Int`: the rank of the used constructor in the `Child`-formula+. It is given value -1 when the constructor is not defined in the input data block,
- `[TypeName]`: the list of types declared for the used node variable in `Child`-formula.

To get the rank and the result type of the used constructor, we must extract it from the `TypeConsTypeArg` given by function *getTypeConsTypeArg*.

- Creating error data based on the list of `IncorrectChild`. If that list is empty, then the formula definition does not violate CSR623. Otherwise, we return the formula name and the list of `IncorrectChild` as error data:

```
ErrDatCSR623 FormulaName [IncorrectChild]
```

CSR624

A `Parent`-formula must fulfill the following conditions:

1. the used constructor must be defined in the input data block,
2. at least one of the constructor's successor types is an element of the declared type of the used node variable (within the `Parent`-quantifier).

To check the above properties, we do as follows:

- extract formula definition information, using *getForInfo*,
- filter the formula structures to contain only `Parent`-formula as subformulas,
- check each `Parent`-formula in the filtered list of formula structures. We create a data type `IncorrectParent` to store error information of each formula. It is defined as follows:

```
data IncorrectParent
  = IncorrectParent FoStruc_Rs [TypeName] [TypeName]
```

  `IncorrectParent` stores the incorrect `Parent`-formula, the successor types of the used constructor, and the declared types of the used node variable.

  We first check whether the used constructor name used is defined in the input data block. If it is defined, then we go on by checking whether there is at least one of the the successor types of the constructor which is an element of the types declared for the used node variable in `Parent`-formula. To know the possible argument types of the constructor, we can extract it from the data type `TypeConsTypeArg`, given by function *getTypeConsTypeArg*. If none of successor types of the constructor is an element of the type declared for the used node variable, then we store that error in `IncorrectParent`.

If the used constructor is not defined, further checking regarding the successor types of the constructor is not needed. We store the error in `IncorrectParent`, by setting the value of both the argument types of the constructor and the types declared for the used node variable to empty list ([]).

- Create error data based on the list of `IncorrectParent`. If that list is empty, then the formula definition does not violate CSR624. Otherwise, we return the formula name and the list of `IncorrectParent` as error data:

```
ErrDatCSR624 FormulaName [IncorrectParent]
```

CSR625

Every subformula used within an mso*-formula must be defined in the same mso block. This checking is done as follows:

- extract formula definition information, using *getForInfo*,
- filter the formula structures to contain only `FS_SUB`-formula as subformulas,
- check whether the formula used in each `FS_SUB`-formula of a formula definition is defined in the same mso block. If yes, then CSR625 is fulfilled. Otherwise, we return the formula name and its list of undefined `FS_SUB`-formulas as error data:

```
ErrDatCSR625 FormulaName [FoStruc_Rs]
```

CSR626

The subformula used within another mso*-formula must get a node variable as argument such that the type of this node variable and the node argument type of the subformula are the same. This restriction is an enhancement of CSR625, since here we only do the checking of argument of formula which is defined in the same mso block, as is required by CSR625. This checking is done as follows:

- the first three steps are just like with CSR625, but we do more checking on the third step. We store the error information of each subformula structure in the data type `IncorrectSub`. It is defined as follows:

```
data IncorrectSub
  = IncorrectSub FoStruc_Rs [TypeName] [TypeName]
```

  `IncorrectSub` stores the incorrect `FS_SUB`-formula, the types of the used node variable, and the declared types for the node argument of the subformula.

  If a subformula is defined, then we go on by checking whether the types of the node variable as argument are the same as the types of the node argument declared for the subformula. If those two types are not the same, then we store that error in `IncorrectSub`. If the subformula name is not defined, then we don't deal with it, because this case has been handled by CSR625.

- Create error data based on the list of `IncorrectSub`. If that list is empty, then the formula definition does not violate CSR626. Otherwise, we return the formula name and the list of `IncorrectSub` as error data:

```
ErrDatCSR626 FormulaName [IncorrectSub]
```

CSR627

We do not allow circular dependencies between formulas in the formula definitions. To check this property, we do as follows:

- extract formula definition information, using *getForInfo*,
- search and collect subformula names which are used in each formula definition recursively. The search will stop when we find one of these two conditions:
  - the root formula name is found in the list of subformulas names to which it depends, or
  - the list of subformula used is exhaustive and there is no more dependency found.

  We store this dependency information in a data type `LinkedDependency`. It is defined as follows:

```
data LinkedDependency
  = LinkedDependency FormulaName [FormulaName]
```

  `FormulaName` stores the root formula name and `[FormulaName]` stores the list of subformula names which is used in the root formula.

- Create error data based on the list of `LinkedDependency`. We check for each formula, whether it has dependency to itself. If it does not have such dependency, then CSR627 is fulfilled. Otherwise, we return the formula name and the list of subformula names to which it depends recursively, as error data:

```
ErrDatCSR627 FormulaName [FormulaName]
```

CSR628

Every formula which uses node or node set variables must use them correctly. It must use a node variable when a node variable is expected. It must use a node set variable when a node set variable is expected. We check this property against formulas which have the possibility to use a node / node set variable. These formulas are contructed by the following constructors of `FoStruc_Rs`:

- `FS_LABEL` (uses one node variable),
- `FS_TYPEOF` (uses one node variable),
- `FS_ELEM` (uses one node variable and one node set variable),
- `FS_EQUAL` (uses two node variables),
- `FS_EDGE` (uses two node variables),

- FS_OVER (uses two node variables),
- FS_UNDER (uses two node variables),
- FS_CHILD (uses one node variable), and
- FS_PARENT (uses one node variable).

Here are the steps to do the checking:

- extract formula definition information, using *getForInfo*,
- check each of the subformulas which uses one of the constructors mentioned above. In general, we must make sure the following conditions hold:
  - If a formula uses a node variable, then:
    - it must be the same as the left-hand side variable, or it is a quantified node variable of an outer quantifier formula.
    - it is not a quantified node set variable of an outer quantifier formula.
  - If a formula uses a node set variable, then:
    - it must be a quantified node set variable of an outer quantifier formula.
    - it is not the same as the left-hand side variable and it is not a quantified node variable of an outer quantifier formula.

  If a formula definition contains subformulas which do not fulfill the above conditions, then we create an error data:

  ```
  ErrDatCSR628 FormulaName [IncorrectUseNV_NSV]
  ```

  The list of `IncorrectUseNV_NSV` stores the error information of each subformula structure. `IncorrectUseNV_NSV` is defined as follows:

  ```
  data IncorrectUseNV_NSV
    = IncorrectUseNV FoStruc_Rs Var1Name
    | IncorrectUseNSV FoStruc_Rs Var2Name
  ```

  `IncorrectUseNV_NSV` has two constructors. The first one is to store error information of a subformula which uses a node variable incorrectly. The second one is to store error information of a subformula which uses a node variable incorrectly.

4.3 Auxiliary Functions

Now we explain about the auxiliary functions. These functions have already been mentioned in the previous subsection when we discuss about the implementation of the restrictions. These functions tend to be general and simple. We explain each of the functions as follows:

1. **getTypeDef**

   collects all formula declarations `TypeDef` from mso block:

   `MsoBlock` $\rightsquigarrow$ *a list of* `TypeDef`

2. **getForDef**

   collects all formula definitions `FormulaDef` from mso block:

   `MsoBlock` ⇝ *a list of* `FormulaDef`

3. **getInputDataBlock**

   gives the data block which is used by mso block as input data block. This function receives a list of data block (`DataEnv`) and the name of the input data block, then returns the corresponding input data block. It is illustrated as follows:

   `DataEnv` *and input definition name* ⇝ *input data block*

4. **getAllTypes**

   collects all the types defined in the input data block. It receives the input data block (given by function *getInputDataBlock*) and then scans through the input data block and returns a list of the types defined in it. The function works in the following order:

   input data block ⇝ *a list of defined types*

5. **getForNodeArg**

   takes a list of formula declarations (given by function *getTypeDef*) and then returns information about the node argument of each formula based on its type signature. This information is realized in a data type `FormulaNodeArg`. This function creates `FormulaNodeArg` in the following scheme:

   `MsoBlock` ⇝ *a list of* `TypeDef` ⇝ *a list of* `TypeSig` ⇝ *a list of* `FormulaNodeArg`

   `FormulaNodeArg` is defined as follows:

   ```
   data FormulaNodeArg
     = FormulaNodeArg FormulaName String [TypeName] TypeRestrOpt
   ```

   Here is an explanation about the elements of data type `FormulaNodeArg`:

   - a constructor `FormulaNodeArg`,
   - `FormulaName`: stores formula name,
   - `String`: stores the type variable when the node argument is a type variable. But this `String` will be empty when the node argument is a type defined in the input data block,
   - `[TypeName]`: stores a list of types when the node argument is a type variable, but this list will be a one-element list when the node argument is a type defined in the input data block,
   - `TypeRestrOpt`: stores the where-from type restriction when the node argument is a type variable. If the node argument is not a type variable, then this element will be set to have the value `TypeRestrOptNull`.

6. **getForInfo**

   receives a list of `Formula Def` and a list of `TypeDef`, given by *getForDef* and *getTypeDef* respectively. Then it returns a list of `FormulaInfo`. This function is illustrated as follows:

   a list of `FormulaDef` *and a list of* `TypeDef` $\rightsquigarrow$ *a list of* `FormulaInfo`

   `FormulaInfo` is defined as follows:

   ```
   data FormulaInfo
     = FormulaInfo FormulaName Var1Name [TypeName] [FormulaVarStruc]
   ```

   `FormulaInfo` consists of the following elements:

   - a constructor `FormulaInfo`,
   - `FormulaName`: stores formula name,
   - `Var1Name`: stores free node variable (on the left-hand side of formula definition),
   - `[TypeName]`: a list of types of the free node variable,
   - `[FormulaVarStruc]`: a list of data type `FormulaVarStruc`. This data type is created by function *getStruc* which we explain after this.

7. **getStruc** receives a right-hand side of formula definition of type `FoStruc_Rs`, the node argument variable, and the list of node argument types, then returns a list of `FormulaVarStruc`. It is illustrated as follows:

   formula `FoStruc_Rs` + *node argument variable* + *a list of node argument types* $\rightsquigarrow$ *a list of* `FormulaVarStruc`

   The definition of `FormulaVarStruc` is given as follows:

   ```
   data FormulaVarStruc
     = FormulaVarStruc
         FoStruc_Rs
         FreeNodeVarList
         BoundedNodeVarList
         InheritedBoundedNodeVarList
         FreeNodeSetVarList
         BoundedNodeSetVarList
         InheritedBoundedNodeSetVarList
         LocalVarList
   ```

   `FormulaVarStruc` contains the following elements:

   - a constructor `FormulaVarStruc`,
   - `FoStruc_Rs`: the current formula,
   - `FreeNodeVarList`: node variables that occur free on a formula in the right-hand side, different from the free node variable on the left-hand side,
   - `BoundedNodeVarList`: node variables that occur bounded for the first time in the right-hand side, which are inside one of the following formulas:

  - FS_EXISTS1 (Var1Name,UpperIdList,(String,String)) FoStruc_Rs,
  - FS_ALL1 (Var1Name,UpperIdList,(String,String)) FoStruc_Rs,
  - FS_CHILD Var1Name ConName Int Var1Name FoStruc_Rs,
  - FS_PARENT Var1Name ConName Var1Name FoStruc_Rs
- InheritedBoundedNodeVarList: node variables that occur bounded by an outer formula, not the current one,
- FreeNodeSetVarList: node set variables that occur free on a formula in the right-hand side, different from the free node variable on the left-hand side,
- BoundedNodeSetVarList: node set variables that occur bounded for the first time in the right-hand side, which are inside one of the following formulas:
  - FS_EXISTS2 (Var2Name,UpperIdList,(String,String)) FoStruc_Rs,
  - FS_ALL2 (Var2Name,UpperIdList,(String,String)) FoStruc_Rs,
- InheritedBoundedNodeSetVarList: node set variables that occur bounded by an outer formula, not the current one,
- LocalVarList: all variables (node or node set) that appear in the right-hand side of formula definition.

The type FreeNodeVarList, BoundedNodeVarList, InheritedBoundedNodeVarList, FreeNodeSetVarList, BoundedNodeSetVarList, InheritedBoundedNodeSetVarList, and LocalVarList are just the renamings for [VarStruc]. We define VarStruc as follows:

```
data VarStruc
  = VarStruc VarName [TypeName]
```

8. **getTypeConsTypeArg**

   receives an input data block, defining the abstract syntax of L, in the form of the data type DataDefList. Then it collects all types, constructors and their successor types by using function *getConsTypeArg* below, and returns them as a list of TypeConsTypeArg. This function is illustrated as follows:

   input data block DataDefList $\rightsquigarrow$ *a list of* TypeConsTypeArg

   The definition of TypeConsTypeArg is as follows:

```
data TypeConsTypeArg
  =  TypeConsTypeArg TypeName [ConsTypeArg]
```

9. **getConsTypeArg**

   receives a non-empty list of constructors defining an algebraic data type, in the form of the data type ConsList, as input. Then collects all constructors and their type arguments by using function *getTypeArg* below, and then returns them as a list of ConsTypeArg. This function is illustrated as follows:

   ConsList $\rightsquigarrow$ *a list of* ConsTypeArg

The definition of `ConsTypeArg` is as follows:

```
data ConsTypeArg
  = ConsTypeArg ConName [TypeName]
```

10. **getTypeArg**

    receives as input type arguments of a constructor, in the form of the data type `TypeList` and then returns a list of type names found in the `TypeList`. This function is illustrated as follows:

    `TypeList` $\rightsquigarrow$ *a list of types*

4.4 User Interface

Now we discuss about the implementation of user interface for the syntax-checker. This user interface takes a list of `ErrDat` as input and then returns clear and friendly error messages about the specification to the specifier. The list of `ErrDat` is collected from the output of every restriction function that we discuss in the previous subsection. By differenciating the unique constructor, the user interface conveniently creates error messages customized for each restriction. The user interface is illustrated as follows:

a list of `ErrDat` $\rightsquigarrow$ *error messages*

5 Conclusions

In this practical work, we implement the syntax-checker for mso*-logic over trees. It integrates all the context-sensitive restrictions into one syntax-checker program. There are totally 28 context-sensitive restrictions. We divide them into three parts: general restrictions (CSR601 - CSR602), the restrictions on formula declarations (CSR603 - CSR609), and the restrictions on formula definitions (CSR610 - CSR628).

The implementation for the general restrictions is quite straightforward. But for the implementation of restrictions on formula declarations, we must first provides information on the type signature of each formula. Based on this information, we do the checking for each restriction. Meanwhile, for the implementation of restrictions on formula definitions, we must work more to provide a well-structured information of formula definitions. The work is to traverse down the tree-structure of every formula definition to collect useful information. This information ranges from the formula name, the node argument of each formula, the hierarchy of subformulas, and the classification of variables. Then based on this information, we check the restrictions for each formula definition.

During the practical work, we discover a new important restriction, which we give the name CSR628. This restriction requires that every formula must use a node or a node set variable in a position where a node or a node set variable is required in the formula. This seemingly simple restriction contributes an important aspect in checking a lot of possible causes of errors of mso*-logic part of DSL specification.

Every restriction function is designed to return rich information about every error that is found. This can be useful in providing more information to the specifier or for other use in the whole generator system.

Finally, future work is required for CSR618 in handling the checking of more complex structure of `Mag`-formulas. So far, we have implemented CSR618 only for the simple structure of `Mag`-formulas. We also note that the syntax-checker has limited information from the parser to check all possible right-hand side structure of `Mag`-formula in the mso*-logic part of DSL specification.

References

[1] E.Bormann. Entwurf und Implementierung eines Systems zur Erzeugung syntaxgesteuerter Editoren. Master's thesis, Dresden University of Technology, 2000.

[2] E. Bormann. Automatic Generation of Language-Based Editors using DSL, 2003. Manuscript.

[3] T. W. Reps and T. Teitelbaum. The Synthesizer Generator. A System for Constructing Language-Based Editors, 1989.

[4] J. H. Fasel, P. Hudak, and J. Peterson. A gentle introduction to Haskell 98. Tutorial, Yale University and University of Carolina, October 1999.

[5] S. Thompson. The Craft of Functional Programming, 1999.