AST V2 Notes

Augusto Ribeiro Kenneth Lausdahl

June 23, 2015

Chapter 1

AST V2

1.1 What is the AST V2?

In order to achieve an ideal common AST three goals have been identified:

1. *Development must be supported both at specification and implementation level, allowing the VDM language itself to be used for tool specification.*
2. *The AST must be extensible while extensions must be kept isolated within a plug-in.*
3. *Sufficient navigation machinery must exist for an editor so that features like e.g. completion and re-factoring can be implemented easily.*

The following subsections will explain the essential principles for the new AST and what changes are required to a generator in order to produce an AST that complies with the identified goals.

It is essential that a new AST is easy to maintain and easy to extend, thus it must only contain functionality essential for the tree structure. To achieve both easy maintenance and support for specification and implementation level development an abstract tree definition can be used like in the `ASTGen` tool. However, extendibility is another matter that need attention; This can be handled by allowing one tree to extend another, by adding new nodes and fields or even refining a type of an existing field.

1.2 AST Creator

probably need to talk about the auxiliary stuff with visitor, etc

The new AST that has been developed has an improved structure compared to both the existing Overture and VDMJ trees. The main addition made here is the ability to extend an AST while keeping the changes isolated in a plug-in architecture. Secondly, the AST is specified using a grammar file inspired by SableCC[1], and can generated to both VDM and Java as supported by `ASTGen`. The main structural change compared to the Overture AST is the ability to add fields to super classes allowing e.g. an expression field to be added to all unary expressions as illustrated in listing 1.1.

Listing 1.1: Extract of the new AST grammar showing the apply and unary expression.

```
Abstract Syntax Tree
exp {-> package='org.overture.ast.expressions'}
    =   {apply} [root]:exp [args]:exp [argtypes]:type*
    |   #Unary
    |   ...
    ;

#Unary {-> package='org.overture.ast.expressions'}
    =  {head}
    |   {tail}
    |   ...

Aspect Declaration
exp->#Unary
    = [exp]:exp;
```

1.3 AST File Structure

An AST file has the following format:

```
Packages
...
Tokens
...
Abstract Syntax Tree
...
Aspect Declaration
...
```

A small description of each section of the AST File is provided below. For more detailed information on each of the AST File sections, consult the correspondent subsection.

[1] `http://www.sablecc.org/`

Packages: in this section it is possible to select the Java packages for the output generated code.

Tokens: in this section it is possible to declare non-generated java classes that can be used as node fields.

Abstract Syntax Tree: this is the section where the AST is declared, composed by nodes and sub-nodes and which fields they contain

Aspect Declaration: In this section it is possible to add fields to all the nodes of a specific type.

1.3.1 Packages

In the "packages" section of the AST file it is possible to select where the package in which the output files of the AST Creator will be located. An example of "packages" section is shown below:

```
Packages
base org.overture.ast.node;
analysis org.overture.ast.analysis;
```

The "base" keyword is used to select where the AST Node Interfaces packages will be put. The "analysis" keyword is the the auxiliary visitors package.

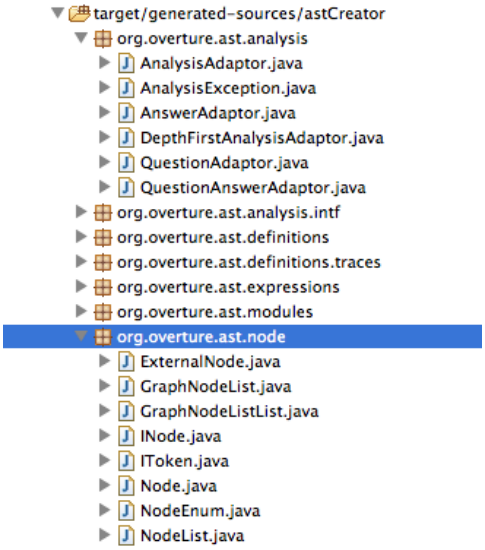


Figure 1.1: The package structure of the generated example.

1.3.2 Tokens

Tokens are non-generated classes which are used by the generated nodes. Tokens can be used as fields of AST Nodes and are Java classes.

There is four different types of token declarations as exemplified below:

```
Tokens

  java_Boolean = 'java:java.lang.Boolean';
  nameScope = 'java:enum:org.overture.ast.typechecker.NameScope';
  LexToken = 'java:node:org.overture.ast.lex.LexToken';
  static = 'static';
```

All of the declaration start by an identifier, but the attribute has some differences in each of them.

java - used to indicate that the token is a Java class.

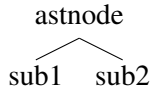
java:enum - used to indicate that the token is a Java enumeration

java:node - used to indicate that the token is a subclass of INode probably need to explain INode in the beginning

string - used to declare a simple token with a content string and location

1.3.3 Abstract Syntax Tree

In this section of the AST file it is where the AST structure is defined.

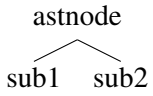


In the following example we want to define an AST with the same structure as the tree presented above.

```
Abstract Syntax Tree
astnode {-> package='org.overture.ast.astNodes'}
    =   {sub1} [root]:astnode [argtypes]:astnode*
    |   {sub2} [exp]:astnode.sub1
    | ...
    ;
```

In this case, the first line defines a node called “astnode”. In the generated output, the ”astnode” will appear prefixed with a **P** (as *PAstnode*). The *package* attribute indicates in which package the classes of the nodes and sub-nodes of this rule will be generated. The generated P-nodes are abstract. It is possible to add fields to the P-node that are then inherited by the sub-nodes, this is explained in **Aspect Declaration**.

class diagram of generated stuff



The following lines declare sub-nodes of P-node, they are the A-nodes. The A-nodes are concrete classes that extend the P-nodes. In the first line of sub-node declaration, the "sub1" node is declared. The "sub1" sub-node name will be generated with the prefix **A** and with its parent node name as suffix (as *ASub1Astnode*). The sub-node "sub1" will contain the fields "root"and "argtypes". If we look a bit closer to the field declaration under "sub1", we can notice different types of declaration.

```
{sub1} [root]:astnode [argtypes]:astnode*
```

The field declaration `[root]:astnode` indicates that the field "root" is of type `astnode`. The field `[argtypes]:astnode*` has a (`*`) after the type of the field, this means this field is a *list* of `astnode`.

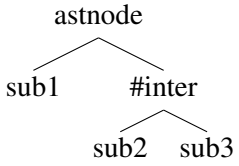
The sub-node "sub2" is declared in the same way as "sub1", but its fields are different.

```
{sub2} [exp]:astnode.sub1
```

It is possible to select the type of the field by using a dot notation if there is the need to point to a specific type. So the node exp on sub-node "sub2" is of type of the sub-node "sub1"

Intermediate Nodes

It is possible to have a tree with more than one level of sub-nodes. We call these intermediate nodes and their names are prefixed with the **S**. S-nodes can be used to create more complex tree structures.



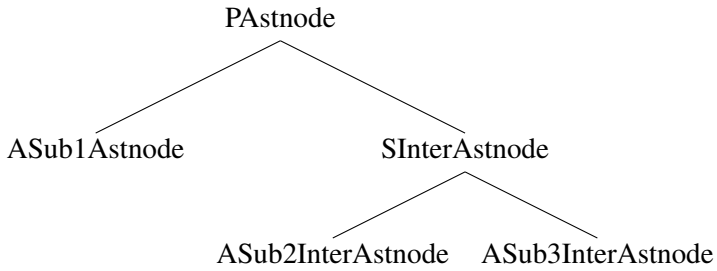

```
Abstract Syntax Tree
astnode {-> package='org.overture.ast.astNodes'}
    =   {sub1} [root]:astnode [argtypes]:astnode*
    |   #inter
    ;

#inter
```

```
= {sub2} [exp]:astnode.sub1
| {sub3} [field]:astnode.#inter
;
```

The output generated from this AST File is the following:

we should put here a class diagram instead]

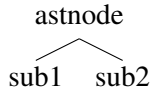


It is possible to define several levels of intermediate nodes.

Graph Nodes

1.3.4 Aspect Declaration

In the last section of the file it is possible do declare fields on P-nodes and S-nodes. This fields will be inherited by all the sub-nodes of the nodes the nodes that declare it. This is a convenient way of adding a field to all sub-nodes of the declaring node.



In the following example we will add a field to “astnode” which is common to all its sub-nodes.

```
Aspect Declaration
%astnode
    = [shared]:java_Boolean
    ;
```

By declaring the field “shared” in “astnode”, “sub1” and “sub2” will inherit this field from .

1.4 Node’s “toString”

The toString macro-processing enables nodes to have a custom toString to make debugging easier. The AST is generated with a toString function that prints all fields one after another. However, in some cases debugging might be eased by printing the node in a more suitable format (e.g. in the syntax of the language at

hand). To allow such customization a macro-processor is available together with a small language.

The base princiles is that all fields of a node is accessed as defined in the AST e.g. field 1 is accessed through [field1] and strings can be added around the fields by a quoted test "example test". The custom toString is defined per node by %node-¿sub1 = and can only be defined for leafs of the tree (**A**-nodes). Furthermore, it is possible to embed java withing the toString by using $ to escape to java. If external java classes are to be used withing the toString any imports that might be used must be specified in the begining of the file with the **`import`** keyword as shown below:

```
To String Extensions
// import packages used by external $$ java code
import org.overture.ast.util.Utils;
import org.overture.ast.util.ToStringUtil;


//  Expressions

%exp->apply = [root] "(" + $Utils.listToString($ [args] $)$ + ")"
```

1.5 Invoking AST Creator using Maven

The AST Creator is made available as a maven plug-in that enables an AST to be automatically build as part of the build process. The current plug-is in avaliable in the overture repo at: `http://build.overturetool.org/builds/mt4e-m2repo-eclipse3.7.2/`

The plug-in is:

groupId	org.overturetool.tools
artifactId	astcreatorplugin
version	1.0.6

The astcreatorplugin plug-in has the goal **generate** for AST generation and the following configuration properties:

deletePackageOnGenerate A list of java packages that should be deleted before generation

ast A path to the ast file starting from src/main/resources

outputDirectory an option to define an alternative output location for the generated AST

The pom configuration is as shown below.

```
<build>
    <plugins>
      <plugin>
        <groupId>org.overturetool.tools</groupId>
        <artifactId>astcreatorplugin</artifactId>
        <version>1.0.6</version>
        <executions>
          <execution>
            <id>java</id>
            <phase>generate-sources</phase>
            <goals>
              <goal>generate</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <deletePackageOnGenerate>
            <param>vdm.generated.node</param>
            <param>org.overture.ast</param>
          </deletePackageOnGenerate>
          <ast>overtureII.astv2</ast>
          <!--outputDirectory>${project.basedir}/src/main/java</outputDirectory -->
        </configuration>
      </plugin>
    </plugins>
```

Chapter 2

AST Extensions

2.1 AST Extension Creator Usage

2.2 Guidelines and Cornercases

To make the use of the AST v2 Extensions more sensible, some rules are defined:

- it is ONLY possible to use ”base visitors” in the extended AST if there is no new nodes in the AST instance. So if ”Base” tree is cloned to ”Extended” tree and the visitor is applied immediately, then it should work. If any ”Extended” node is added this visitor will not work and throw a ”RuntimeException” a an unknown node is hit.
- Calling the ”base” kind method of an ”extended” node will result in a ”RuntimeException”
- The generated ”Extended” visitor will contain the new cases for the added nodes, the cases of the nodes that already existed but were changed in the extension (added fields) and will not contain the ones that are unchanged.
- Setting the ”Base” parent in the ”Extended” tree will result in a ”RuntimeException”
- When cloning a tree, extra care should be taken to preserve the graph fields connections. A solution might be to clone the tree first without graph fields and keep a map of ”object $\rightarrow$ cloned object” and in the second pass fill the graph fields gap.
- If an extra field was added to a node existing in the base case the visitor case needs to overwritten in the extended visitor.

- It is not advised to mix nodes from base tree with the extended tree.

Appendix A

Class diagrams of the Extended AST

A.1 Base

A.2 Extended

A.3 Extended AST Test Class

```
import junit.framework.TestCase;

import org.overture.ast.expressions.AE1Exp;
import org.overture.ast.expressions.AE1ExpInterpreter;
import org.overture.ast.expressions.AE2ExpInterpreter;
import org.overture.ast.expressions.AE3Exp;
import org.overture.ast.expressions.AE3ExpInterpreter;
import org.overture.ast.expressions.AE4ExpInterpreter;
import org.overture.ast.expressions.EExp;
import org.overture.ast.expressions.EExpInterpreter;
import org.overture.ast.node.NodeEnum;
import org.overture.ast.node.NodeEnumInterpreter;

public class TestCase1 extends TestCase
{
  AE1Exp base = null;
  AE2ExpInterpreter extended = null;

  @Override
  protected void setUp() throws Exception
  {
    base = new AE1Exp();
```

```
  extended = new AE2ExpInterpreter();
}

public void testenum()
{
  base.kindNode();
  extended.kindNode();

  base.kindPExp();
  try
  {
    extended.kindPExp();
    fail("kindPExp successeded but should have failed");
  } catch (RuntimeException e)
  {

  }

  extended.kindPExpInterpreter();
}

public void testBaseVisitor()
{
  System.out.println("testBaseVisitor");
  base.apply(new BaseVisitor());

  try
  {
    extended.apply(new BaseVisitor());
    fail("apply with BaseVisitor successeded but should have failed since this exp is not
  } catch (RuntimeException e)
  {

  }

}

@SuppressWarnings("deprecation")
public void testAddedFieldToNode()
{

  System.out.println("Base visit E3");
  AE1Exp field1Base = new AE1Exp();
  AE3Exp rootBase = new AE3Exp(field1Base);
  DBaseVisitor vBase = new DBaseVisitor();
  rootBase.apply(vBase);
  assertTrue("Root must be visited first", vBase.visitedNodes.get(0).equals(rootBase));
  assertTrue("Field 1 of root must be visited second", vBase.visitedNodes.get(1).equals(f
  assertEquals(vBase.visitedNodes.size(), 2);
```

```
  // System.out.println("Mixed visit E3");
  // AE1ExpInterpreter field1_ = new AE1ExpInterpreter();
  // AE3Exp mixtedRoot = new AE3Exp(field1_);
  // try
  // {
  // mixtedRoot.apply(vBase);
  // fail("E1 interpreter can not be visited with a base visitor. To allow this the exter
  // } catch (RuntimeException e)
  // {
  //
  // }
  //
  // System.out.println("Mixed Extended visit E3");
  // DExtendedDelegateVisitor extendedDelegateVisitor = new DExtendedDelegateVisitor();
  // mixtedRoot.apply(extendedDelegateVisitor);
  // System.out.println(extendedDelegateVisitor.visitedNodes);

  System.out.println("Extended visit E3");
  AE1ExpInterpreter field1_ = new AE1ExpInterpreter();
  AE2ExpInterpreter field2_ = extended;
  AE3ExpInterpreter root = new AE3ExpInterpreter(field1_, field2_);

  DExtendedVisitor vExtended = new DExtendedVisitor();
  root.apply(vExtended);

  assertTrue("Root must be visited first", vExtended.visitedNodes.get(0).equals(root));
  assertTrue("Field 1 of root must be visited second", vExtended.visitedNodes.get(1).equa
  assertTrue("Field 2 of root must be visited second", vExtended.visitedNodes.get(2).equa
  System.out.println(vExtended.visitedNodes);
}

public void testExtendVisitor()
{
  System.out.println("testExtendVisitor");
  base.apply(new ExtendedVisitor());
  extended.apply(new ExtendedVisitor());

}

public void testPExtendedEnums()
{
  for (NodeEnum e1 : NodeEnum.values())
  {
    try
    {
      Enum.valueOf(NodeEnumInterpreter.class, e1.toString());
    } catch (IllegalArgumentException e)
    {
```

```
        fail(e1 + " should exist in both ASTs");
      }
    }

    for (EExp e1 : EExp.values())
    {
      try
      {
        Enum.valueOf(EExpInterpreter.class, e1.toString());
      } catch (IllegalArgumentException e)
      {
        fail(e1 + " should exist in both ASTs");
      }
    }

    try
    {
      if (Enum.valueOf(NodeEnum.class, NodeEnumInterpreter.STM.toString()) != null)
      {
        fail("STM should exist in both ASTs");
      }
      fail("STM should exist in both ASTs");
    } catch (IllegalArgumentException e)
    {

    }

    try
    {
      if (Enum.valueOf(EExp.class, EExpInterpreter.E2.toString()) != null)
      {
        fail("E2 should exist in both ASTs");
      }
    } catch (IllegalArgumentException e)
    {

    }
  }

  @SuppressWarnings("deprecation")
  public void testclone()
  {
    AE2ExpInterpreter a=  extended.clone();
    System.out.println(a);

    AE1ExpInterpreter field1_ = new AE1ExpInterpreter();
    AE2ExpInterpreter field2_ = extended;
    AE3ExpInterpreter root = new AE3ExpInterpreter(field1_, field2_);
```

```
  AE3ExpInterpreter  r2 = root.clone();
  if(r2==root)
  {
    fail("Clone did not work");
  }

  if(r2.getField1()==field1_)
  {
    fail("Clone did not work");
  }

  if(r2.getField2()==field2_)
  {
    fail("Clone did not work");
  }
}

@SuppressWarnings("deprecation")
public void testParent()
{
  AE1ExpInterpreter field1_ = new AE1ExpInterpreter();
  AE2ExpInterpreter field2_ = extended;
  AE3ExpInterpreter root = new AE3ExpInterpreter(field1_, field2_);
  if(field1_.parent()!=root)
  {
    fail("Parent not set correctly");
  }
  if(field2_.parent()!=root)
  {
    fail("Parent not set correctly");
  }
  if(root.parent()!=null)
  {
    fail("Parent not set correctly");
  }

  AE3ExpInterpreter root2 = new AE3ExpInterpreter(field1_, field2_);

  if(field1_.parent()!=root2)
  {
    fail("Parent not set correctly");
  }
  if(field2_.parent()!=root2)
  {
    fail("Parent not set correctly");
  }
  if(root2.parent()!=null)
  {
    fail("Parent not set correctly");
```

```
    }
  }


  @SuppressWarnings("deprecation")
  public void testGraphField()
  {
    AE1ExpInterpreter field1_ = new AE1ExpInterpreter();
    if(field1_.parent()!=null)
    {
      fail("Parent not set correctly");
    }
    AE4ExpInterpreter e4 = new AE4ExpInterpreter(field1_);
    if(field1_.parent()!=e4)
    {
      fail("Parent not set correctly");
    }

    AE4ExpInterpreter e4_1 = new AE4ExpInterpreter(field1_);

    if(field1_.parent()!=e4)
    {
      fail("Parent not set correctly");
    }

    AE3ExpInterpreter e3 = new AE3ExpInterpreter(field1_, null);

    if(field1_.parent()!=e3)
    {
      fail("Parent not set correctly");
    }

    if(e4.getField2()==null || e4_1.getField2()==null)
    {
      fail("Graph nodes should not be removed when used in other nodes");
    }

  }
}
```

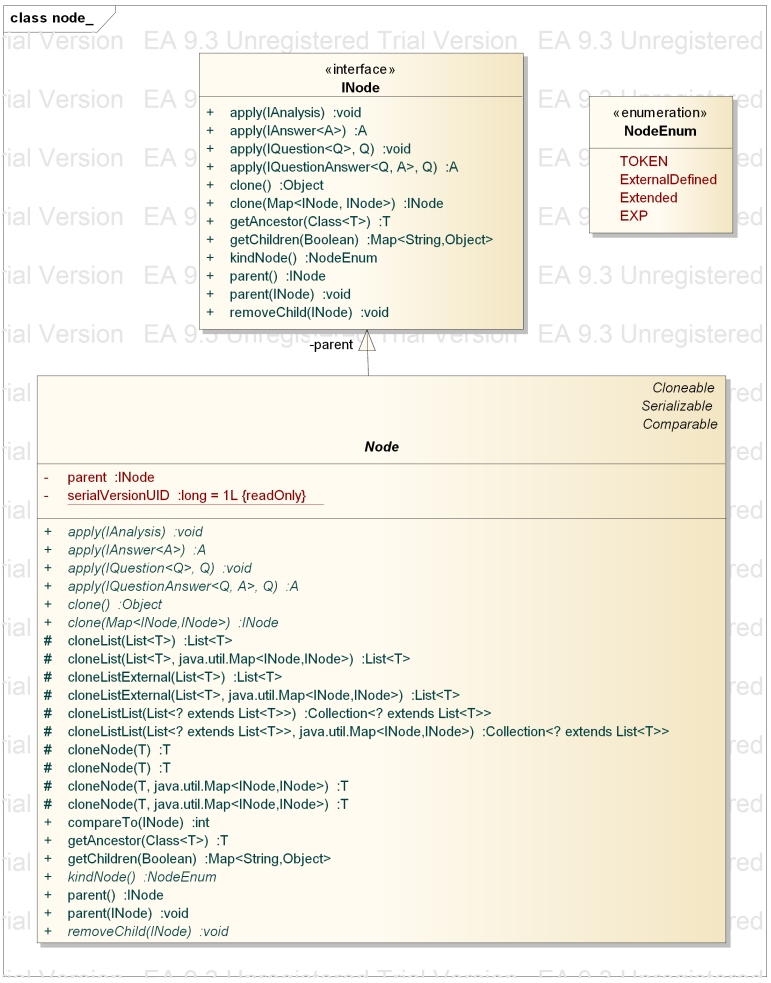


Figure A.1: Base node package

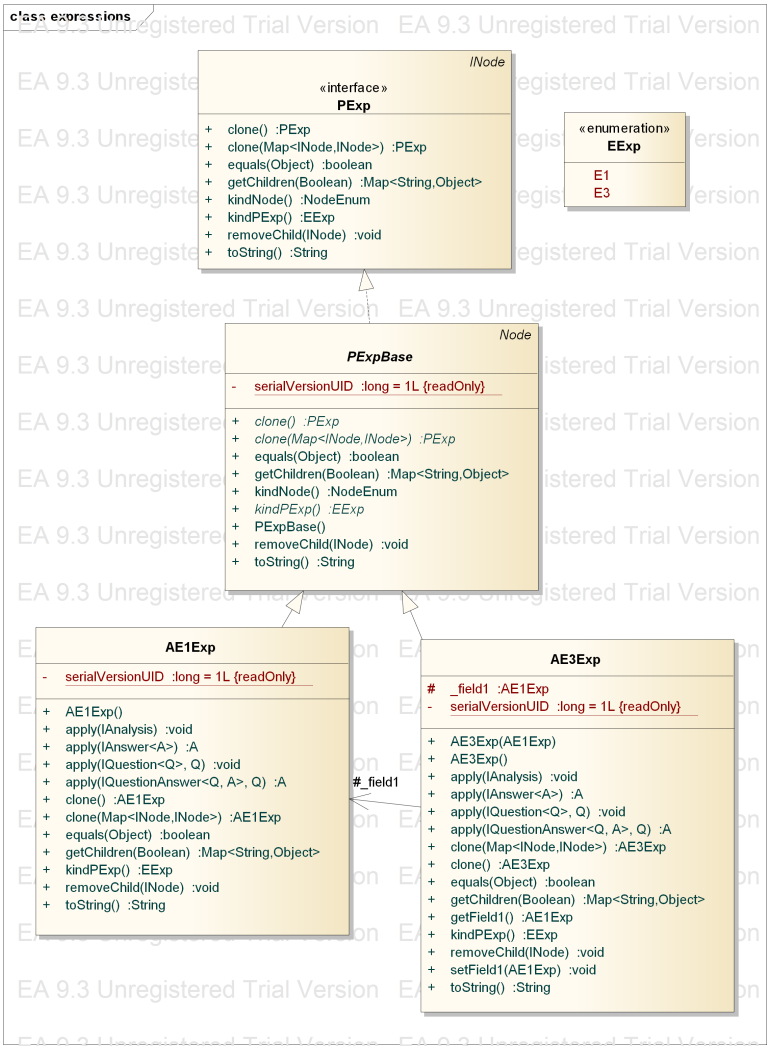


Figure A.2: Base expression package

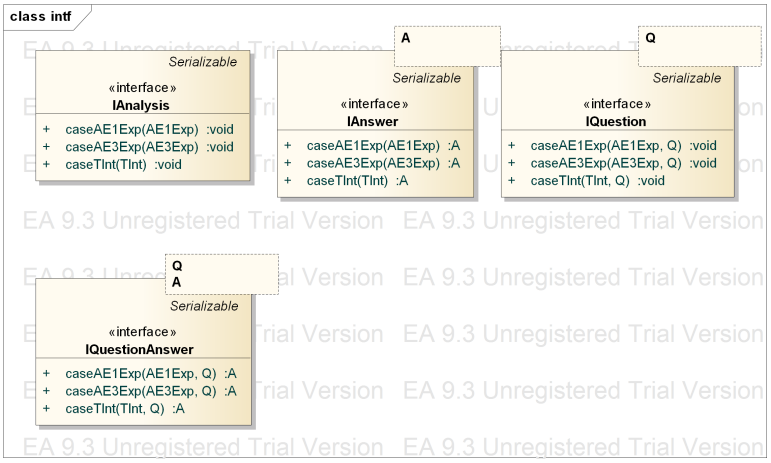


Figure A.3: Base analysis interfaces package

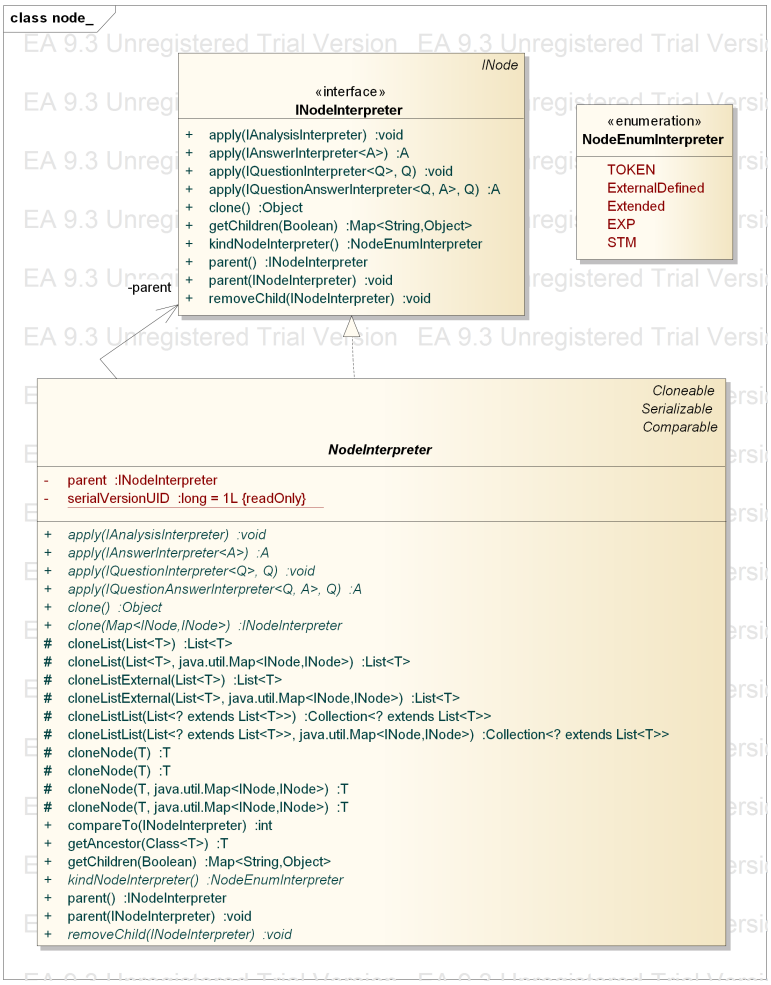


Figure A.4: Extended node package

Figure A.5: Extended expressions package

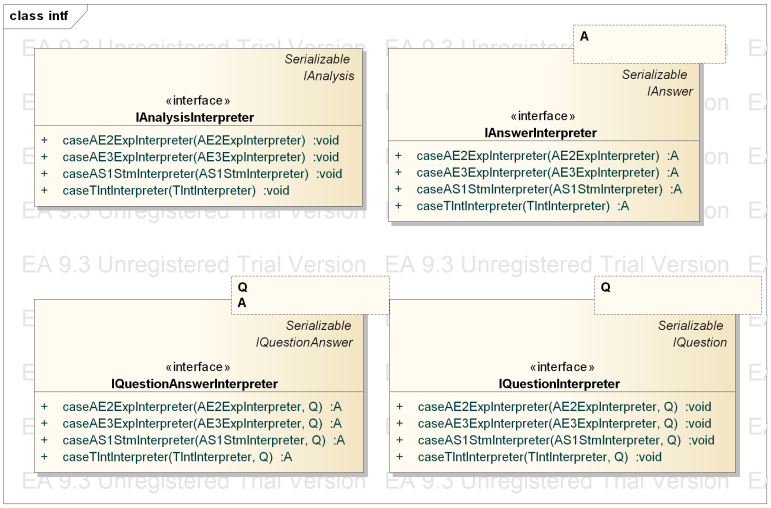


Figure A.6: Extended analysis interfaces package

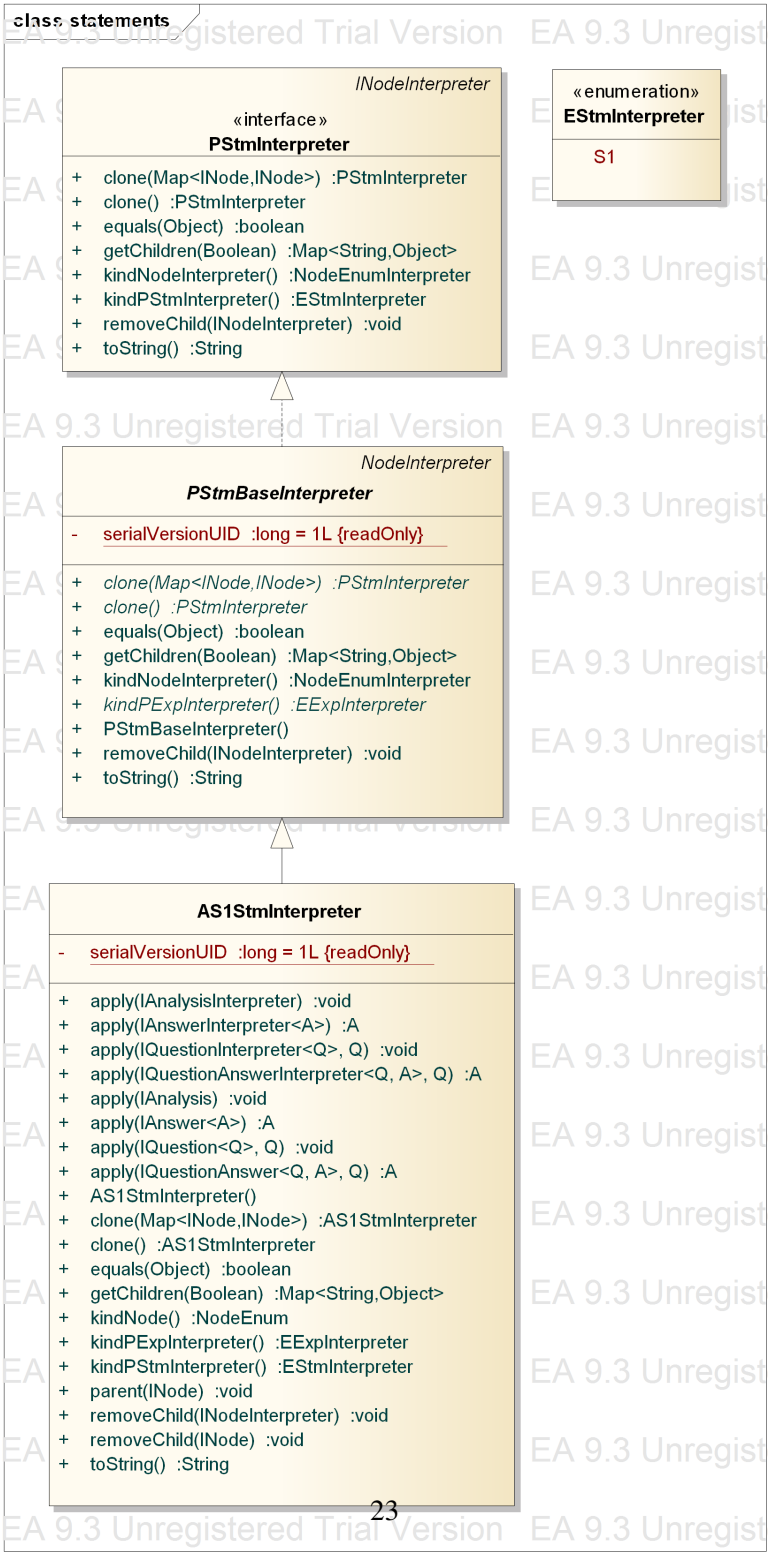


Figure A.7: Extended statements package

Appendix B

Challanges

- Tree consistency
- Tree automated analysis; visit in decent order
- Generating a tree to replace OO developed classed. Dificult because of functionality embedded in the classes where 1,2,4 super classes all might override some methods.
- Caching: converting a program which does OO caching/linking to use a generated tree. Not easy, solutions:
  - Hashmap
  - Add custum graph fields to the tree. Nodes in a graph field would be different because they are not a child of the node when are attached to. In relation to tree fields which can only have one parent and thus needs to disconnect from their previously attached parent before moving to a new,
- Handling OO embeded functions outside a tree. Developing assistants to handle public/private functions from OO tree. The difficulty is to keep the hierarchy intact.
- Display: For debugging it is important to have a decent toString of a tree, two solutions:
  - Generated toString including all fields of a node and it name.
  - Custom toString. Since a tree might represent (to some extend) the syntax of a language a toString representation close to the real syntax might be prefered. One solution would be to add a way to define the

toString method of nodes based on their fields and strings plus possibly allowing external java methods to be called for more advanced display functions.