AST...

K.G. Lausdahl and Augusto Ribeiro

Aarhus School of Engineering, Dalgas Avenue 2, DK-8000 Aarhus C, Denmark

Abstract. ...

Keywords: VDM

1 Introduction

1.1 Challenges

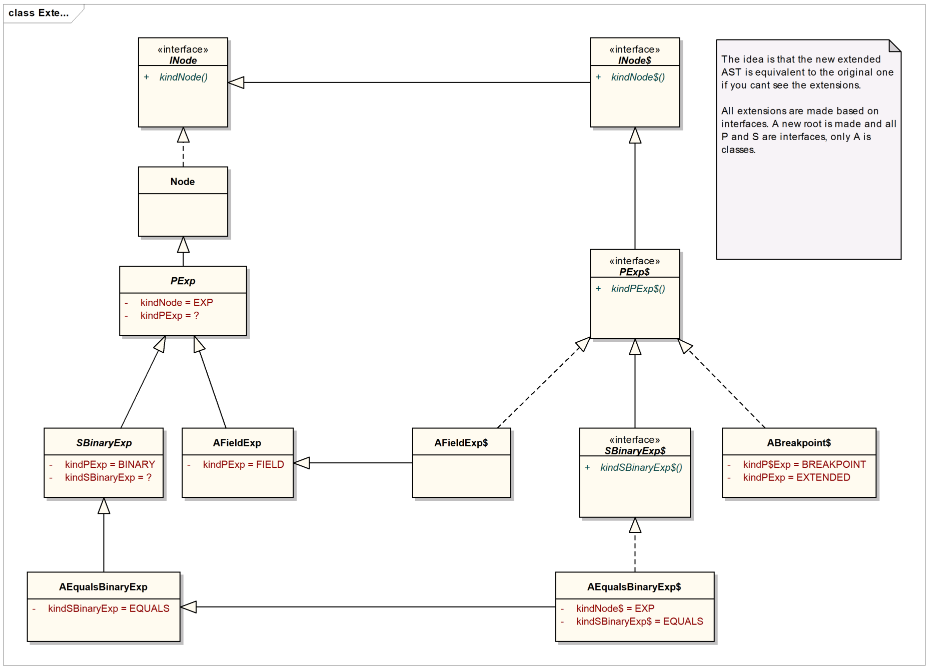
- Tree consistency
- Tree automated analysis; visit in decent order
- Generating a tree to replace OO developed classed. Dificult because of functionality embedded in the classes where 1,2,4 super classes all might override some methods.
- Caching: converting a program which does OO caching/linking to 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.

1.2 Extension

1.3 Editors

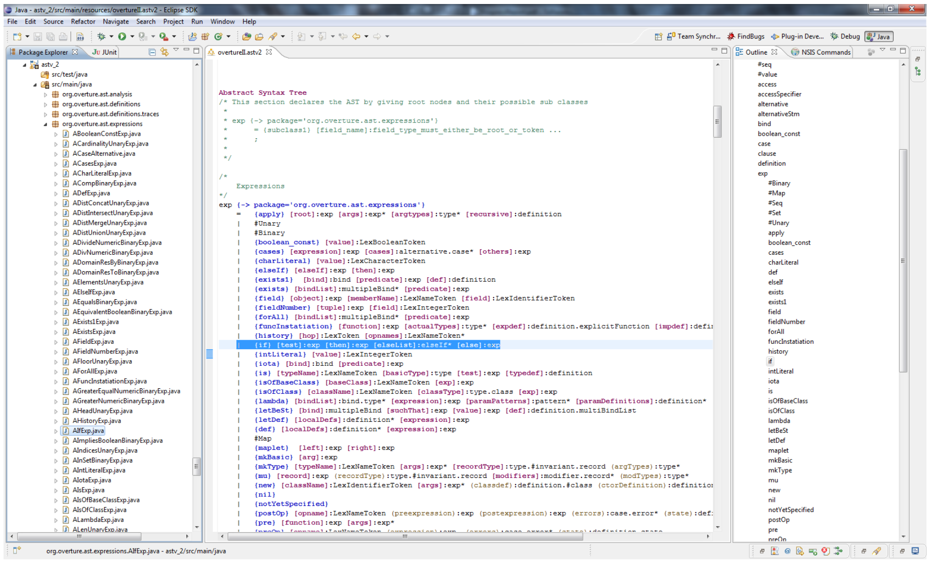
2 Abstract Syntax Tree Generation

The motivation for this work is based on the experience gained through the Overture projects developments history. Through the last 10 years multiple tools has been developed by different people in different places where many of the tools has their own



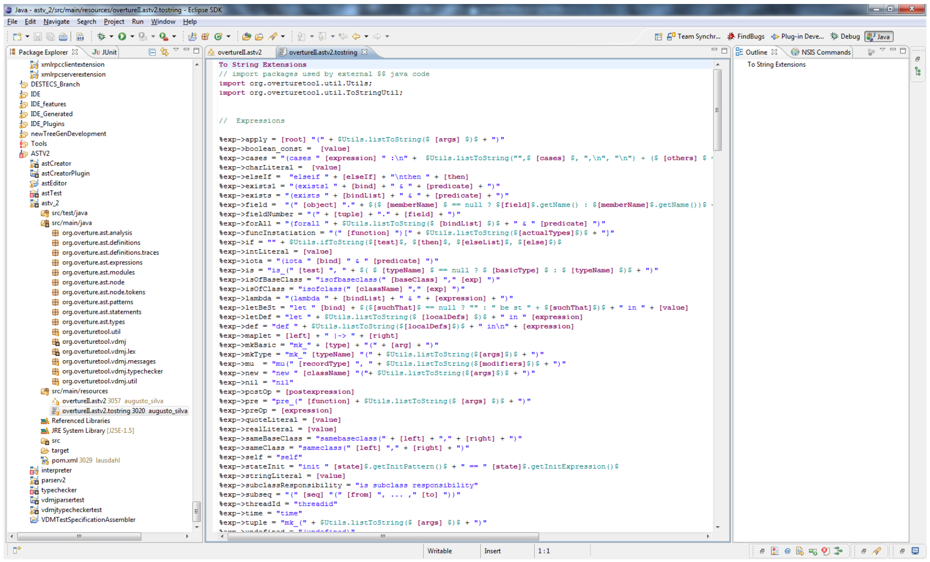
The idea is that the new extended AST is equivalent to the original one if you cant see the extensions.

All extensions are made based on interfaces. A new root is made and all P and S are interfaces, only A is classes.



```
Abstract Syntax Tree
/* This section declares the AST by giving root nodes and their possible sub classes
 *
 * exp {-> package='org.overture.ast.expressions'}
 *      = {subclass1} [field_name]:field_type_must_either_be_root_or_token ...
 *      ;
 *
 */

/*
    Expressions
*/
exp {-> package='org.overture.ast.expressions'}
    =   {apply} [root]:exp [args]:exp* [argtypes]:type* [recursive]:definition
    |   #Unary
    |   #Binary
    |   {boolean_const} [value]:LexBooleanToken
    |   {cases} [expression]:exp [cases]:alternative.case* [others]:exp
    |   {charLiteral} [value]:LexCharacterToken
    |   {elseIf} [elseIf]:exp [then]:exp
    |   {exists1}  [bind]:bind [predicate]:exp [def]:definition
    |   {exists} [bindList]:multipleBind* [predicate]:exp
    |   {field} [object]:exp [memberName]:LexNameToken [field]:LexIdentifierToken
    |   {fieldNumber} [tuple]:exp [field]:LexIntegerToken
    |   {forAll} [bindList]:multipleBind* [predicate]:exp
    |   {funcInstatiation} [function]:exp [actualTypes]:type* [expdef]:definition.explicitFunction [impdef]:defi
    |   {history} [hop]:LexToken [opnames]:LexNameToken*
    |   {if} [test]:exp [then]:exp [elseList]:elseIf* [else]:exp
    |   {intLiteral} [value]:LexIntegerToken
    |   {iota} [bind]:bind [predicate]:exp
    |   {is} [typeName]:LexNameToken [basicType]:type [test]:exp [typedef]:definition
    |   {isOfBaseClass} [baseClass]:LexNameToken [exp]:exp
    |   {isOfClass} [className]:LexNameToken [classType]:type.class [exp]:exp
    |   {lambda} [bindList]:bind.type* [expression]:exp [paramPatterns]:pattern* [paramDefinitions]:definition*
    |   {letBeSt} [bind]:multipleBind [suchThat]:exp [value]:exp [def]:definition.multiBindList
    |   {letDef} [localDefs]:definition* [expression]:exp
    |   {def} [localDefs]:definition* [expression]:exp
    |   #Map
    |   {maplet}  [left]:exp [right]:exp
    |   {mkBasic} [arg]:exp
    |   {mkType} [typeName]:LexNameToken [args]:exp* [recordType]:type.#invariant.record (argTypes):type*
    |   {mu} [record]:exp (recordType):type.#invariant.record [modifiers]:modifier.record* (modTypes):type*
    |   {new} [className]:LexIdentifierToken [args]:exp* (classdef):definition.#class (ctorDefinition):definitio
    |   {nil}
    |   {notYetSpecified}
    |   {postOp} [opname]:LexNameToken (preexpression):exp (postexpression):exp (errors):case.error* (state):def
    |   {pre} [function]:exp [args]:exp*
    |   {preOp} [opname]:LexNameToken (expression):exp  (errors):case.error* (state):definition.state
```


Java - astv_2/src/main/resources/overtureII.astv2.tostring - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Package Explorer
JUnit
overtureII.astv2
overtureII.astv2.tostring
Outline
NSIS Commands
To String Extensions
Writable
Insert
1:1

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


//  Expressions

%exp->apply = [root] "(" + $Utils.listToString($ [args] $)$ + ")"
%exp->boolean_const =  [value]
%exp->cases = "(cases " [expression] " :\n" +  $Utils.listToString("",$ [cases] $, ",\n", "\n") + ($ [others] $
%exp->charLiteral =  [value]
%exp->elseIf =  "elseif " + [elseIf] + "\nthen " + [then]
%exp->exists1 = "(exists1 " + [bind] + " & " + [predicate] + ")"
%exp->exists = "(exists " + [bindList] + " & " + [predicate] + ")"
%exp->field =  "(" [object] "." + $($ [memberName] $ == null ? $[field]$.getName() : $[memberName]$.getName())$
%exp->fieldNumber = "(" + [tuple] + "." + [field] + ")"
%exp->forAll = "(forall " + $Utils.listToString($ [bindList] $)$ + " & " [predicate] ")"
%exp->funcInstatiation = "(" [function] ")[" + $Utils.listToString($[actualTypes]$)$ + "]"
%exp->if = "" + $Utils.ifToString($[test]$, $[then]$, $[elseList]$, $[else]$)$
%exp->intLiteral = [value]
%exp->iota = "(iota " [bind] " & " [predicate] ")"
%exp->is = "is_(" [test] ", " + $( $ [typeName] $ == null ? $ [basicType] $ : $ [typeName] $)$ + ")"
%exp->isOfBaseClass = "isofbaseclass(" [baseClass] "," [exp] ")"
%exp->isOfClass = "isofclass(" [className] "," [exp] ")"
%exp->lambda = "(lambda " + [bindList] + " & " + [expression] + ")"
%exp->letBeSt = "let " [bind] + $($[suchThat]$ == null ? "" : " be st " + $[suchThat]$)$ + " in " + [value]
%exp->letDef = "let " + $Utils.listToString($ [localDefs] $)$ + " in " [expression]
%exp->def = "def " + $Utils.listToString($[localDefs]$)$ + " in\n" + [expression]
%exp->maplet = [left] + " |-> " + [right]
%exp->mkBasic = "mk_" + [type] + "(" + [arg] + ")"
%exp->mkType = "mk_" [typeName] "(" + $Utils.listToString($[args]$)$ + ")"
%exp->mu  = "mu(" [recordType] ", " + $Utils.listToString($[modifiers]$)$ + ")"
%exp->new = "new " [className] "("+ $Utils.listToString($[args]$)$ + ")"
%exp->nil = "nil"
%exp->postOp = [postexpression]
%exp->pre = "pre_(" [function] + $Utils.listToString($ [args] $)$ + ")"
%exp->preOp = [expression]
%exp->quoteLiteral = [value]
%exp->realLiteral = [value]
%exp->sameBaseClass = "samebaseclass(" + [left] + "," + [right] + ")"
%exp->sameClass = "sameclass(" [left] "," + [right] + ")"
%exp->self = "self"
%exp->stateInit = "init " [state]$.getInitPattern()$ + " == " [state]$.getInitExpression()$
%exp->stringLiteral = [value]
%exp->subclassResponsibility = "is subclass responsibility"
%exp->subseq = "(" [seq] "(" [from] ", ... ," [to] "))"
%exp->threadId = "threadid"
%exp->time = "time"
%exp->tuple = "mk_(" + $Utils.listToString($ [args] $)$ + ")"
%exp->undefined = "(undefined)"
```

version of an AST which is almost the same structure vice. To create some AST which all tools could use and which would be easy to extend allowing each tool to add its own custom fields a new approach was needed. If a new AST should be made a few of the main points where it should be improved would be:

- Defining the AST through a single grammar.
- Be able to generate a source code for both Java and VDM. Allowing both development at specification and code level.
- Support of a range of visitors allowing an easy way to visit e.g. any expression no matter where it is used in the AST.
- An easy way to extend a existing AST with additional fields on a node and adding new nodes.
- An easy way to specify a toString like functionality for the nodes allowing a customized representation of each node to be used during debugging.

In figure 1 the main idea is shown where a gramme file defining the AST structure is given to the generator which in turn will output a number of classes for both Java and VDM defining the structure.

Before any work was carried out the current the AST which existed was investigated. Here it turned out that the oldest Overture AST was infarct build on the concept of generating a tree structure based on a grammar file and with visitor support, however the structure of the output was not an OO structure where e.g. all expressions had a common base class. If compared to the AST used by the current interpreter which was a good quality OO structure which also had good debugging support with proper toString methods allowing an easy way to display nodes to the developer reducing the effort required to understand the structure. However this second structure didnt have any visitor support meaning that any attempt to visit e.g. an expression would reacquire

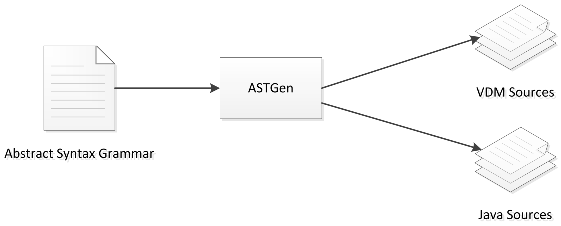


Fig. 1: Overview the AST generator.

the developer to hand write an entire visitor for the complete tree. Through in-depth analysis and prototyping of an AST generator a few challenges was identified in the process of changing a handwritten AST into a generated one. The handwritten AST had a normal hierarchy where each node had its own implementation of a `typeCheck` and `eval` method both of which must be moved out into an visiting structure. This in it self did not cause any major issues however all small utility methods defined in each node which was overridden in only some of the nodes did. It required a large number of assistants switching on the type. Another important discovery was that for optimization the handwritten AST was not a pure tree structure infarct many of the childs of a node was pointers to other nodes in the AST populated during type checking where e.g. the definition of a node was associated with the node for later use. This caused a problem in relation to the original idea where each node only could be associated to one other node. To overcome this a custom type of childs for a node was introduced named graph fields which did not remove the childs parent when set.

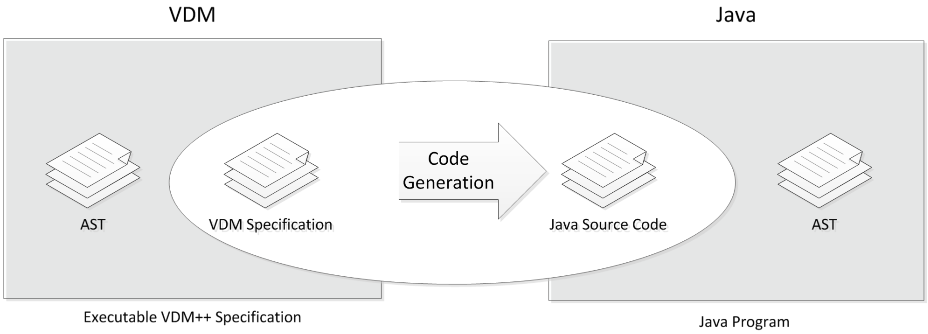


Fig. 2: Flow for AST Sources to Java program.

2.1 Extending an existing AST

For the interpreter, extend with new expression breakpoint and adding breakpoint to all expressions as a field.

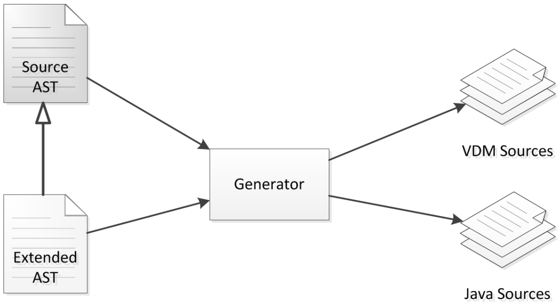


Fig. 3: Extending an existing AST.

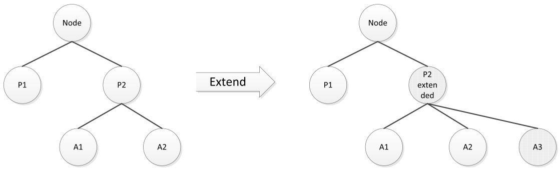


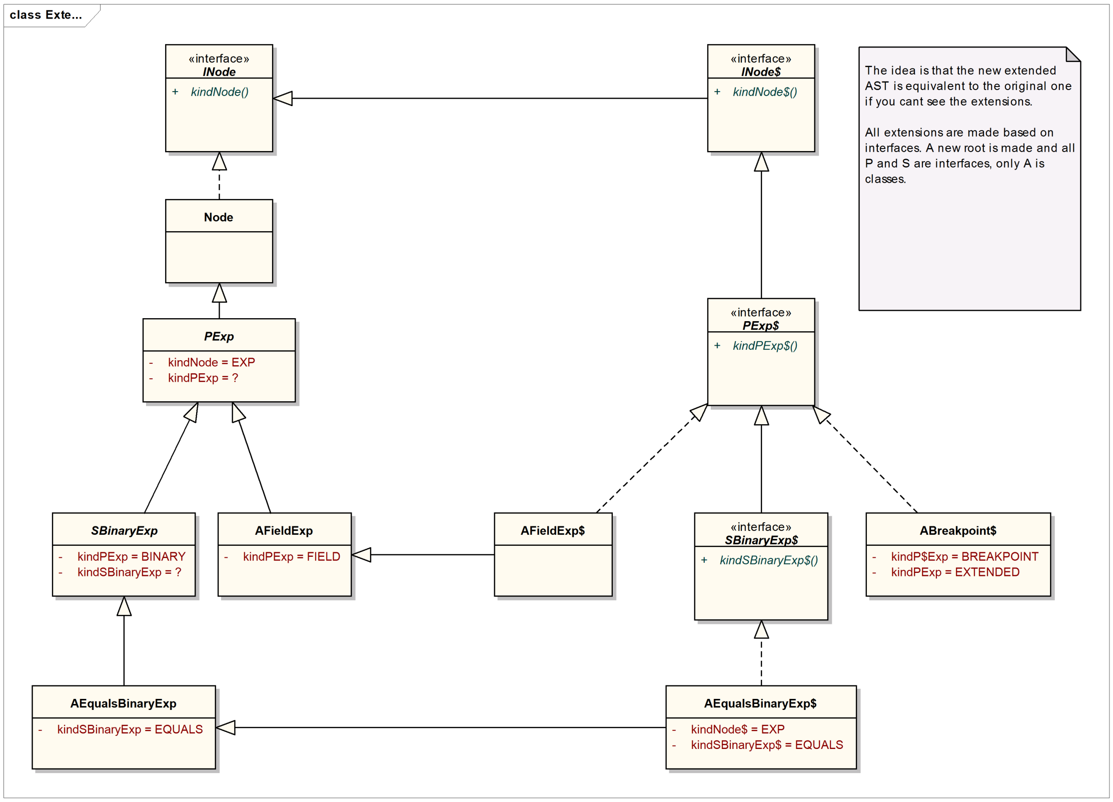
Fig. 4: Extending an existing AST.

2.2 Challenges - this section is just notes

Goals

- Have one AST definition on a simple grammar like format
- Be able to get sources for both Java and VDM
- Have a way to automatically visit nodes of the tree in depended of in what structure they are used.
- Extend an existing tree without the need to specify more then the new additions while still being able to use the tree as the source tree (type vice)
- Have an AST with a toString debugging feature which can be customized allowing a better representation of the nodes.

- Tree consistency
- Tree automated analysis; visit in decent order
- Generating a tree to replace OO developed classed. Difficult 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,



The idea is that the new extended AST is equivalent to the original one if you cant see the extensions.

All extensions are made based on interfaces. A new root is made and all P and S are interfaces, only A is classes.

Fig. 5: AST preview of extended hierarchy.

- 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.