Comment- and Indentation Preserving Refactoring and Unparsing for Modelica

Peter Fritzson, Adrian Pop, Kristoffer Norling, Mikael Blom
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, SE-581 83 Linköping, Sweden
{petfr, adrpo, x06krino, x06mikbl}@ida.liu.se

Abstract

In this paper we present a strategy for comment- and indentation preserving refactoring and unparsing for Modelica. The approach is general, but is currently being implemented for Modelica in the OpenModelica environment. We believe this to be one of the first unparsing approaches that can preserve all user-defined indentation and comment information, as well as fulfilling the principle of minimal replacement at refactorings.

Keywords: Refactoring, comments, unparsing,, Modelica.

1 Introduction

Integrated programming environments, e.g. InterLisp [11] and Eclipse [12] provide various degrees of support for program transformations intended to improve the structure of programs – so-called *refactorings* [5] (see also Section 10).

Such operations typically operate on abstract syntax tree (AST) representations of the program. Therefore the program needs to be converted to tree form by *parsing* before refactoring, and be converted back into text by the process of *unparsing*, also called *pretty printing* This is supported by a number of environments (Section 10).

However, a well-known problem is that of preserving comments and user-defined indentation while performing refactorings. Essentially all current environments either loose the comments (except for special comments that are part of the language syntax and AST representation), or move them to some other place. User-defined indentation is typically lost and replaced by machine-generated standard indentations. This is accepted by some developers, but judged as unacceptable by others. However, if the objective only is to improve indentation, then a semi-automatic indenter can be used instead (Section 8.3).

Currently Modelica-based tools are handling only declaration comments that are part of the model and are discarding or moving all the other comments, i.e. the ones between `/* */` and after `//….` Such behavior is highly undesirable from a user perspective and heavily affects the ease-of-use of code-versioning tools.

A goal for the work presented here is to support Modelica code refactoring with minimal disruption of user-defined comments and indentation. In this paper we present such an approach for unparsing in conjunction with refactorings.

2 Comments and Indentation

Regard the following contrived Modelica example. It has one declaration comment which is part of the language syntax, and two “textual” comments `Itemcomm` and `MyComm` which would be eliminated by a conventional parser. It is also nicely hand formatted so that the start positions of each component name in the text are vertically aligned.

```
record MODIFICATION  "Declaration comment"

  Boolean            finalItem; //Itemcomm
  Each /* MyComm */  eachRef;
  ComponentRef       componentReg;

end MODIFICATION;
```

Assume that this is parsed and unparsed by a conventional (comment-preserving) unparser, putting two blanks between the type and the component name of each component. The manual indentation would be lost, and the “textual” comments would be moved to some standard positions (or be lost):

```
record MODIFICATION "Declaration comment"

  Boolean  finalItem; //Itemcomm
  Each  eachRef; /* MyComm */
  ComponentRef  componentReg;

end MODIFICATION;
```

3 Refactorings

Below we make some general observations and give examples of refactorings.

3.1 The Principle of Minimal Replacement

For a refactoring to have minimal disruption on the existing code, it is desired that it supports the principle of minimal replacement:

- *When replacing a subtree, the minimal subtree that contains the change should be replaced.*

This also has the consequence of minimal loss or change of comments. For example, if a name (an identifier) is changed, only the identifier node in the tree should be replaced, not the surrounding subtree.

3.2 Some Examples of Refactorings

Here we mention a few common refactorings. There are also numerous, more advanced and specialized refactorings.

- *Component name change*. Change name of a component name in a record. For example:

```
record MODIFICATION  "Declaration comment"
  Boolean            finalItem; //Itemcomm
  Each /* MyComm */  eachRef;
  ComponentRef       componentReg;
end MODIFICATION;
```

  The name of the component reference name is currently `componentReg`, which is an error. It should be `componentRef`. We would like to change the name both in the declaration and all its uses, thus avoiding updating all named references by hand, which would be quite tedious.

- *Function name change*. Change the name of a function, both the declaration and all call sites.

- *Add record component*. Add a new component declaration to record. In MetaModelica, that would also mean putting an underscore '_' at the correct position in all patterns for that record type with positional matching.

- *Add function formal parameter*. Add an input or output formal parameter to a function. The question is, how much is possible to do automatically? Adding arguments to recursive calls to the function itself is no great problem, but calls from other functions can be more problematic since meaningful input data needs to be provided. This can be handled easily in those cases a default value can be passed to the function's new formal parameter.

4 Representing Comments and User-Defined Indentation

How should information about comments and user defined indentation be represented in the internal (AST) program representation? There are basically two possibilities for a chunk of code, e.g. a model:

- *Tree*. The AST representation is the main storage (the TRUTH). Comments and indentation as extra nodes/attributes in the AST.
- *Text*. The text representation, including indentation and comments, is the main storage (the TRUTH).

The tree approach may seem natural, since the refactorings and the compiler operate on the tree representation. However, it has some disadvantages:

- Since white space and comments can appear essentially anywhere, between nodes, associated with nodes, the AST will become cluttered and increase the required memory usage and complexity of the tree, perhaps by a factor 2-3.
- The large number of extra nodes in the AST may complicate code accessing and traversing the tree.

Regarding the text representation we make the following observations:

- The text representation exists from the start, since this is the storage form used in the file system. Environments like Eclipse use text buffers for direct interaction with the programmer.
- The text representation includes all indentation and comment information, and is compact.
- The structure of the program in the text representation is not apparent, and cannot be easily manipulated.

Why not combine the advantages of each representation, and try to avoid the disadvantages?

- Use the text representation as the basic storage format including indentation and comment information. The text might be conceptually divided into chunks, where for example each class definition gives rise to a text chunk.
- Use the tree representation for compilation and refactoring. Create it when needed and keep it during the current session. Create it piece-wise, e.g. for one class at a time.
- Create a mapping from the tree representation to the text representation; each node in the tree has a corresponding position and size in the text representation. Create this mapping when needed, for appropriate pieces (e.g. class definitions) of the total model.

5 Implementation

The following strategy is used for the implementation

5.1 Base Program representation

The text representation is the TRUTH, the source, and the AST representation is a secondary representation derived from the source, used during compilation and refactoring.

The class information attribute of a class definition in the AST should be extended, e.g. with the byte start position (directly addressing within a file), or by a text chunk corresponding to the text of a class declaration. A package which contains classes would instead refer to the definitions of those classes.

Text positions and text sizes of each AST node should be indirectly associated with each AST node.

5.2 The Parser

The following special considerations need to be addressed by the parser:

- In order not to clutter the produced AST tree, the parser produces two trees: a standard AST tree, and a positioning tree (produced in parallel) with the same number of nodes, containing text positions and sizes of each subtree.
- The parser should return the start text position and text size of each built AST tree. Moreover, if there are any comments within the AST tree text range, a list of the start positions and sizes of these comments should be associated with the parallel tree node.
- The pure AST tree should be clean and not cluttered with position and comment information.
- As mentioned, a text position tree with the same number of nodes and children as the AST is created in parallel to the AST. The positioning tree is only produced when needed for refactorings or text positioning, and thrown away when not needed.

For example, a child nr 3 of a node at level 2, will find its text positions in the parallel tree in the node at level 2 and child nr 3.

5.3 The Scanner

The text position and size of each token is returned together with the token itself.

5.4 The New Unparser

The new unparser will use a combined strategy as follows, combining existing text with new text generated by the tree unparser:

- If there exist already indented text associated with a node, use this text to produce the unparsing text.
- If there is no existing text, this must be a new tree node produced by the refactoring tool. Call the tree unparser to convert this subtree into text that is inserted into the final unparsing result.

6 Refactoring Process

The following steps are to performed in this order during the actual refactoring:

- Traverse the AST and perform insertion/deletion/replacement of subtrees.
- For each insertion/deletion/replacement operation, put each such an operation descriptor in a list, together with the text position and size of the text of the subtree to be replaced/deleted etc.
- After traversal, sort these operations according to text position, and perform the operations in the text in backwards order (take those at the highest text position first).

7 Example of Function Name Refactoring

The example below is used to illustrate the refactorings and the used combined tree and text chunk representation.

All loaded models (including the `Modelica` package) reside in an un-named top-level scope that we can call `Top`. A model may be a top-level model, but more typically a package which in turn may consist of subpackages:

```
01 within ParentPackage;
02 package pack
03  function addOne "function that adds 1"
04   input Real x = 1.0; // line comment
05   output Real y;      /*  multiple
06                           line
07                           comment */
08  algorithm
09   y := x + 1.0;
10  end addOne;
11
12  class myClass
13    Real y;
14  equation
15    y = addOne(5); // Call to addOne
16  end myClass;
17 end pack;
```

Line numbers are given to help the reader follow the example. The position tree constructed by the parser is given in the appendix as it is quite large. A portion of the abstract syntax tree is also shown in order to understand the example.

A function name refactoring will be applied to the example which will change the name of the function `"addOne"` to `"add1"`, The refactoring can be performed in the OpenModelica environment by loading the example and calling the interactive API function:

```
loadFileForRefactoring("Example.mo");
refactorFunctionName(pack.addOne, "add1");
```

The compiler will execute the first command by calling the new parser that also builds the position tree together with the AST:

```
(ast,posTree) = Parse.refactorParse(file);
```

The result of the load command is two trees. The second (`posTree`) is the position tree presented (partly) in the appendix. The first (`ast`) is the abstract syntax tree of the loaded file which is presented also in the appendix entirely. Here is just a overview picture of the AST:

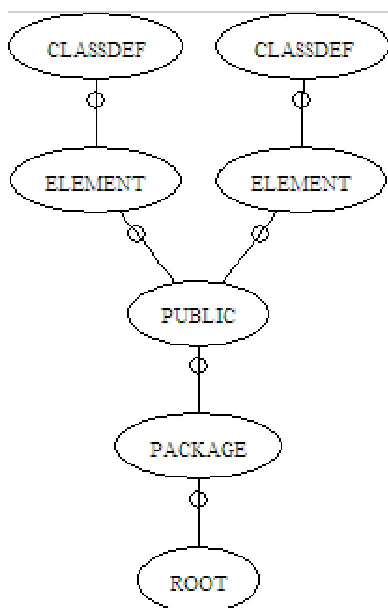

Figure 1. AST of the Example.mo file.

The figure shows that the program has one package with two public elements which are class definitions.

Actually only two refactoring operations are needed to implement any refactoring: add and delete or add and replace.

When `refactorFunctionName` is called the compiler will perform these operations:

7.1 Lookup pack.addOne

Lookup of a class definition is performed by walking the AST while keeping track of a numbered path in the tree. To reach the `addOne` identifier, the path: 1, 6, 1, 1, 1, 5, 2, 1, 1 is applied. The path goes via the following AST nodes: `PROGRAM [1] / CLASS [6] / PARTS [1] / PUBLIC [1] / ELEMENTITEM [1] / ELEMENT [5] / CLASSDEF [2] / CLASS [1] / IDENT("addOne") [1]`.

7.2 Lookup Any Uses of pack.addOne

Lookup of the uses are performed by walking the AST, keeping track of the scope, while keeping track of a numbered path. To reach the function call of `addOne`, the path: 1, 6, 1, 1, 1, 5, 2, 1, 1 is applied. The path goes via the following AST nodes:

`PROGRAM [1] / CLASS [6] / PARTS [1] / PUBLIC [2] / ELEMENTITEM [1] / ELEMENT [5] / CLASSDEF [2] / CLASS [6] / PARTS[1] / EQUATIONS [1] / EQUATIONITEM [1] / EQ_EQUALS [2] / CALL[1] / CREF_IDENT [1] / IDENT("addOne") [1]`.

7.3 Apply the Refactoring to the Actual Text

Now that the paths needed for the minimal refactoring were discovered in the AST, apply these paths to the position tree and fetch the positions of the elements at the end of the paths:

- Function name: `IDENT, Start:047, End:053`
- Function use: `IDENT, Start:313, End:319`

The text operations are applied bottom-up because otherwise the character positions of the elements below an applied operation would change. Ordering of text operations is needed to have them applied in a bottom-up fashion:

- `ReplaceText(file, 319, 313, "add1");`
- `ReplaceText(file, 53, 47, "add1");`
- `Close(file);`
- `(ast, posTree) =  // re-parse the file Parse.refactorParse(file);`

After the file is closed either a reparsing is performed to load the new AST (as exemplified here) or the refactoring operations are perfomed on the tree already in the memory. Of course the best alternative would be to perform the refactoring during lookup as we have implemented it in the OpenModelica compiler.

As one can notice the comments stay in place so there is minimal disruption to the text representation. This is very valuable from a user point of view but also for code-versioning tools.

7.4 Calculation of the Additional Overhead

There is not too much overhead for the refactoring both with respect to memory usage and time spent walking the tree. In the following table we discuss such overhead and give specific numbers for needed memory size and time complexity of the refactoring procedure.

Memory overhead	Time overhead
Space is required for storing the position tree. The size of this space is two integers (of 4 bytes) for each AST node. Also the list of operations to be applied to the text needs memory for storing the paths and the operations themselves, but this memory is negligible compared to the AST and position tree and can also be freed. **Example**: there are about 50 nodes in the example, which means an additional memory of ~ 50NrNodes x 2Positions x 4Bytes = 400Bytes are needed for the position tree. Or course, the position tree could be built on demand and the freed when memory is needed.	Walking two trees while performing the refactoring has a time impact of *NumberOfNodesWalked* x O(1) to walk a node: O(*NrOfNodesWalked*). Walking the position tree while and applying the text operations to the file is negligible compared to the refactoring operation. **Example**: it took about 0.2 seconds to perform the function name refactoring for the example file using the OpenModelica system. Refactoring old graphical annotations of the Modelica Standard Library version 1.6 to the new style graphical annotations took about 9.6 seconds, which is very good for such a demanding refactoring.

8 Unparsers/Prettyprinters versus Indenters

As mentioned previously, an unparser converts an AST program representation into (nicely indented) text. A reformatting indentation tool uses another approach, it operates directly on the text representation to produce a more nicely indented text.

8.1 Pretty printers/Unparser Generators

An unparser generator produces an unparser from a specification, a grammar-like description of unparsing related aspects of the language. A number of systems mentioned in Section 8 support unparsing or generation of unparsers from such specifications.

8.2 OpenModelica Tree Unparser

The current OpenModelica version 1.4 unparser is hand implemented in MetaModelica, recursively traversing the AST while generating the Modelica text representation. It can be invoked by the OpenModelica `list` command. Comments are currently lost (except for declaration comments).

8.3 Reformatting Indentation in the OpenModelica Eclipse Plugin

A *text reformatting indentation* tool operates directly on the text representation, and analyzes the text by a combination of scanning and piecemeal heuristic partial parsing to recognize certain combinations of tokens. It inserts or removes white space in order to produce a nice indentation, or improve an existing one. Such mechanisms are typically invoked by the user on a few lines at a time, and are not completely automatic, the user is often required to perform the final adjustments. An advantage with this approach is that comments are not lost.

This kind of indentation tool is for example available for a number of languages in their respective Emacs modes, or as part of Eclipse plugins, e.g. for C++, Java, and more recently for Modelica in the OpenModelica MDT Eclipse plugin.

MDT includes support for automatic indentation, as described here and in [13]. When typing the Return (Enter) key, the next line is indented correctly. The user can also correct indentation of the current line or a range selection using CTRL+I or "Correct Indentation" action on the toolbar or in the Edit menu.

Indentation can be applied to incomplete code as a heuristic Modelica scanner is used and the indentation is based only on the tokens generated by this scanner. The indenter indents one line at a time. For example, consider that line four (4) in Figure 2 should be indented. The indenter asks the heuristic scanner to give tokens from the starting token in backwards direction to the start of the file until a scope introducer is recognized, which for this particular file is `model MoonAndEarth`. The reference position of the start of the scope introducer is computed and line four (4) is indented from this reference position one indent unit. The indentation result is presented in Figure 2.

Indenting Modelica code is far from trivial when incomplete (possibly incorrect) code should be indented correctly. Most of the difficulty comes from Modelica scopes which are hard to recognize using just a scanner and some logic behind it. In languages like C/C++ and Java finding enclosing scopes is very easy as one character tokens are used for the scope opening and closing: `"{"` and `"}"`. In Modelica you need at least two tokens and much more case analysis to find where a scope starts and ends. Complications also arise when mixing if-statements with if-expressions (which was further complicated by the introduction of conditional declarations in the Modelica language). In this particular case we implemented a parser emulator that recognizes these constructs based on scanner tokens delivered backwards.

```
model MoonAndEarth
  import Modelica.Math.sin;
  import Modelica.Math.cos;
  Body earth(
    x(start=0),
    y(start=0),
    vx(start=0),
    vy(start=0),
    mass=5.9736*10^24);
  Body moon(
    x(start=384400000),
    y(start=0),
    vx(start=0),
    vy(start=1023.14),
    mass=7.347673*10^22);
  constant Real G=6.67300*10^(-11) "gravitational constant";
  Real force;
  Real xDistance;
  Real yDistance;
  Real alfa;
equation
  xDistance = moon.x - earth.x;
  yDistance = moon.y - earth.y;
  cos(alfa) = xDistance/sqrt(xDistance^2 + yDistance^2);
```

Figure 2. Example of code after automatic indentation.

The indenter works well in almost all cases, but there are cases in which is impossible to find the correct indentation. For example when the indentation of a line consisting of `"end Name;"` is requested and the scope introducer for `Name` is not found (that is identifier `Name` followed backwards by `class, model, package, block, record, connector` etc.) then the indenter fails and returns the indentation of the previous line.

9 Further Discussion

In this section we address some questions from the reviewers:

Question: "A question I have always had is whether there are any "mistakes" in the grammar that should be corrected with respect to these issues. Similarly, how is this handled with the Java tools in Eclipse?"

Answer: The answer to this question highly depends on the syntactic mistake the user made. For example if an **"end if;"** is missing at the end of an equation section, but is followed by **"end** `Model;`**"**, then such a mistake can be automatically corrected using a heuristic parser. However, if an opening scope is missing, i.e., **model** `Model` (or alternatively an ending scope) there is no way to know where it should be introduced. There are a lot of places that can be proposed:

- Just after the enclosing scope starts (after i.e., `package MyPack` introduction) if there exists such scope or the start of the file if no such scope exists.
- Just after the every existing ending scope of a model found by going backwards from the **end** `Model;`

Right now the Eclipse environment will call the OpenModelica compiler to parse the file each time the file is saved. The parsing errors are reported in the Eclipse environment as a list of errors, but also underlined where the error occurs as shown in Figure 3. Of course if the user selects an entire file and calls the automatic indentation routine, the indentation will work correctly if there are no *large large grammatical errors in the file*.

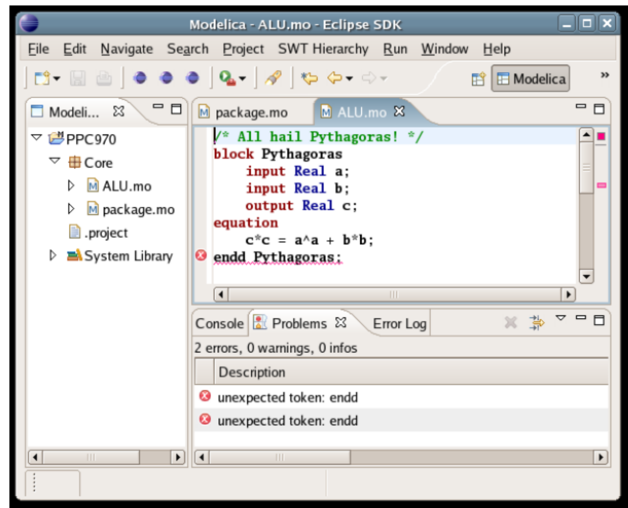

Figure 3. Syntax checking.

Question: "Dymola's pretty printing algorithm does not appear to be deterministic (it sometimes changes files for no reason just because they have been re-saved). Please discuss this deterministic issue and also what implications the algorithms will have for version control tools (i.e. avoiding complex or unnecessary changes since this will complicate "merge" operations)."

Answer: As exemplified in Sections 3.1 and 7 the disruption to the actual text is minimal so the code-versioning tools would have no problem with merging operations. This was one of our goals when designing and implementing the refactoring tools presented in the paper. The algorithms in this paper also apply to Modelica models constructed programmatically because these can also be viewed as refactorings. In general the construction of models programmatically is performed by a visual component diagram editor. The editor will give commands: `addModel(…)`, `addComponent(…)`, `addConnection(…)`, etc., to the internal handler of the textual model (that works on the AST and the positionTree) which in the case of a file with code formatting will minimally disrupt the existing code and add all the new code correctly indented at the end or in other appropriate places.

10 Related Work

The term refactoring and its use in a general and systematic sense was introduced by Martin Fowler et al [5], also based on earlier work, even though similar

code transformation operations were previously available, e.g. in the InterLisp environment [11].

Early work in interactive integrated programming environments including unparsing/pretty printing supporting a specific language was done in the InterLisp system for the Lisp language [11], common principles and experience of early interactive Lisp environments are described in [16], a generic editor/unparser/parser generator used for Pascal (and later Ada) in the DICE system [9], [10], the integrated Mjölner environment with mullti-language editing and unparsing support [17]. None of these approaches preserve comments when unparsing, except the InterLisp environment where the comments were already part of the AST which was just pretty printed with a more readable indentation. However, also in the InterLisp case, all hand indentation and white space added by the user is lost, and text style comments (not part of the AST) are also lost.

Many parser generation systems, e.g. ANTLR [14], Eli [6], CoCo [15], also support unparsing from the generated AST, but do not support preservation of comments and hand-made indentation.

11 Conclusions

We have given a preliminary description of refactorings together with an approach for comment- and indentation preserving unparsing. This is currently ongoing work. Part of the unparser and the refactorings are implemented. A full prototype implementation is expected to be completed early spring 2008.

12 Acknowledgements

This work has been supported by the Swedish Foundation for Strategic Research (SSF) in the RISE and VISIMOD projects, by Vinnova in the Safe and Secure Modeling and Simulation project, and by the Swedish Research Council (VR).

References

[1] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe*, 44/45, Dec 2005. http://ww.ida.liu.se/projects/OpenModelica

[2] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 940 pp., Wiley-IEEE Press, 2004.

[3] Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proc. of the 4th International Modelica Conference*, Hamburg, Germany, March 7-8, 2005.

[4] The Modelica Association. *The Modelica Language Specification Version 3.0*, September 2007. http://www.modelica.org.

[5] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, June 1999.

[6] Uwe Kastens, William M. Waite, and Anthony M. Sloane,. *Generating Software from Specifications*. ISBN 0763741248. Jones and Bartlett Publishers. 2007.

[7] William W Pugh; Steven J Sinofsky. A new language-independent prettyprinting algorithm. Ithaca, NY : Dept. of Computer Science, Cornell University, 1987.

[8] Martin Mikelsons. Prettyprinting in an interactive programming environment. In *Proc. of ACM SIGPLAN SIGOA symposium on Text manipulation*. Portland, Oregon, 1981.

[9] Peter Fritzson. *Towards a Distributed Programming Environment based on Incremental Compilation*. 161 pages. PhD thesis no 109, Linköping University, April 13 1984.

[10] Peter Fritzson. Symbolic Debugging through Incremental Compilation in an Integrated Environment, *Journal of Systems and Software*, 3, pp. 285–294, 1983.

[11] Teitelman, Warren. *INTERLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, CA, 1974.

[12] Eclipse website. http://www.eclipse.org. Referenced Nov 2007.

[13] Adrian Pop, Peter Fritzson, Andreas Remar, Elmir Jagudin, and David Akhvlediani. OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging. In *Proceedings of the 5th International Modelica Conference (Modelica'2006)*, Vienna, Austria, Sept. 4-5, 2006.

[14] http://www.antlr.org. ANTLR. Accessed Nov 2007.

[15] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß. The Compiler Generator Coco/R. http://www.ssw.uni-linz.ac.at/coco/. Accessed Nov 2007.

[16] Erik Sandewall. Programming in an Interactive Environment: The "LISP" Experience, *Computing Surveys*, 10:1, Mar. 1978.

[17] J. Lindskov, M. Knudsen, O. Löfgren, Ole Lehrmann-Madsen, and Boris Magnusson (Eds.). *Object-Oriented Environments - The Mjølner Approach*. Prentice Hall, 1993.

Appendix

Here we give (parts of) the generated position tree (posAST) for the code in the example section. The start and end are given in character offsets. The nodes that have -1 as start/end position do not actually exist in the text, but they appear in here to have 1-to-1 mapping to the AST definitions.

```
(Program, (Start: 1, End: 366, {
 (list<Class>, (Start: 23, End: 366, {
   (Class, (Start: 23, End: 366, { (Ident, (Start: 31, End: 35)
     (Boolean Partial, (Start: -1, End: -1) (Boolean Final, (Start: -1, End: -1)
     (Boolen Ecapsulated, (Start: -1, End: -1) (Restriction, (Start: 23, End: 30)
     (ClassDef, (Start: 35, End: 356, {
       (list<ClassPart>, (Start: 38, End: 356, {
         (ClassPart, (Start: 38, End: 356, {
           (list<ElementItem>, (Start: 38, End: 356, {
             (ElementItem, (Start: 38, End: 264, {
               (Element, (Start: 38, End: 264, {
                 (Boolean final, (Start: -1, End: -1)
                 (Option<RedeclareKeywords>, (Start: -1, End: -1)
                 (InnerOuter, (Start: -1, End: -1)
                 (Ident, (Start: -1, End: -1)
                 (ElementSpecEL5, (Start: 38, End: 264, {
                   (Boolean replaceable, (Start: -1, End: -1)
                   (Class, (Start: 53, End: 264, {
                     (Ident, (Start: 47, End: 53)
                     (Boolean Partial, (Start: -1, End: -1)
                     (Boolean Final, (Start: -1, End: -1)
                     (Boolen Ecapsulated, (Start: -1, End: -1)
                     (Restriction, (Start: 38, End: 46)
                     (ClassDef, (Start: 53, End: 264, {
                       (list<ClassPart>, (Start: 53, End: 264, {
                         (ClassPart, (Start: 80, End: 250, {
                           (list<ElementItem>, (Start: 80, End: 221, {
                             (ElementItem, (Start: 80, End: 100, {
                               (Element, (Start: 80, End: 100, {
                                 (Boolean final, (Start: -1, End: -1)
                                 (Option<RedeclareKeywords>,(Start: -1, End: -1)
                                 (InnerOuter, (Start: -1, End: -1)
                                 (Ident, (Start: 91, End: 92)
                                 (ElementSpecEL3, (Start: 91, End: 100, {
                                   (ElementAttributes, (Start: 80, End: 85, {
                                     (Boolean flow, (Start: -1, End: -1)
                                     (Variability, (Start: -1, End: -1)
                                     (Direction, (Start: 80, End: 85)
                                     (ArrayDim, (Start: -1, End: -1)
                                   })
                                   (TypeSpec, (Start: 86, End: 90, {
                                     (Path, (Start: 86, End: 90, {
                                       (Ident, (Start: 86, End: 90)
                                     })
                                     (Option<ArrayDim>, (Start: -1, End: -1)
                                   })
       ... // truncated text due to its large size
       }) (Option<String>, (Start: -1, End: -1)
     }) (Info, (Start: -1, End: -1)
   })
 })
 (Within, (Start: 1, End: 7,
  (Path, (Start: 8, End: 22, {(Ident, (Start: 8, End: 22)})
)
```

Here is another version of the example with character positions for end and start of a Modelica construct:

```
[001]within[007] [008]ParentPackage;[022]
[023]package[030] [031]pack[035]
[036]  [038]function[046] [047]addOne[053] [054]"function that adds 1"[076]
[077]   [080]input[085] [086]Real[090] [091]x[092] [093]=[094] [095]1.0;[099]
                             [100]// line comment[115]
[116]   [119]output[125] [126]Real[130] [131]y;[133]
                             [139]/*  multiple
                                      line
                                      comment */[221]
[222]  [224]algorithm[233]
[234]   [237]y[238] [239]:=[241] [242]x[243] [244]+[245] [246]1.0;[250]
[251]  [253]end[256] [257]addOne;[264]
[265]
[266]  [268]class[273] [274]myClass[281]
[282]    [286]Real[290] [291]y;[293]
[294]  [296]equation[304]
[305]    [309]y[310] [311]=[312] [313]addOne[319](5);[323] [324]// Call to addOne[341]
[342]  [344]end[347] [348]myClass;[356]
[357]end[360] [361]pack;[366]
```

Parts of the abstract syntax tree (AST) of the Example.mo in the example section is presented below. The AST has exactly the same structure as the position tree.

```
adrpo@KAFKA /c/home/adrpo/doc/projects/modelica2008/
$ omc +d=dump Example.mo
Absyn.PROGRAM([
 Absyn.CLASS(Absyn.IDENT("pack"),
  false, false, false, Absyn.R_PACKAGE,
  Absyn.PARTS(
   [Absyn.PUBLIC(
     [Absyn.ELEMENTITEM(
       Absyn.ELEMENT(false, _, Absyn.UNSPECIFIED , "function",
         Absyn.CLASSDEF(false,
          Absyn.CLASS(Absyn.IDENT("addOne"),
           false, false, false, Absyn.R_FUNCTION,
           Absyn.PARTS(
            [Absyn.PUBLIC(
             [Absyn.ELEMENTITEM(
               Absyn.ELEMENT(false, _, Absyn.UNSPECIFIED, "comp",
                Absyn.COMPONENTS(Absyn.ATTR(false, Absyn.VAR, Absyn.INPUT,[]),
                 Absyn.PATH(Absyn.IDENT("Real")),
                 [Absyn.COMPONENTITEM(
                   Absyn.COMPONENT(Absyn.IDENT("x"),[],
                    SOME(Absyn.CLASSMOD([], SOME(Absyn.REAL(1.0))))), NONE)]),
                Absyn.INFO("Example.mo", false, 4, 4, 4, 22)), NONE)),
              Absyn.ELEMENTITEM(
               Absyn.ELEMENT(false, _, Absyn.UNSPECIFIED , "component",
                Absyn.COMPONENTS(Absyn.ATTR(false, Absyn.VAR, Absyn.OUTPUT, []),
                 Absyn.PATH(Absyn.IDENT("Real")),
                 [Absyn.COMPONENTITEM(Absyn.COMPONENT("y",[],
                   NONE), NONE)]),
                Absyn.INFO("Example.mo", false, 5, 4, 5, 17)), NONE))]),
            Absyn.ALGORITHMS(
             ALGORITHMITEM(
              ALG_ASSIGN(
               Absyn.CREF(Absyn.CREF_IDENT("y", [])),
                Absyn.BINARY(
                 Absyn.CREF(Absyn.CREF_IDENT("x", [])),
                 Absyn.ADD,
                 Absyn.REAL(1.0))))))],
            SOME("function that adds 1")),
            Absyn.INFO("Example.mo", false, 3, 3, 10, 13))
  ... // truncated text due to its large size
 ], // end of Absyn.CLASS list
 Absyn.WITHIN(Absyn.IDENT("ParentPackage")
) // end Absyn.PROGRAM
```